Exhibit B

Samsung Pay Developers
Onboarding and Project Integration Guide

In-App Payments for Merchants

Portal v1.0 | SDK v1.8.00 | Doc Rev 3.1-US | November 2017

SAMSUNG pay

PLEASE READ

The content and specifications contained in this document supersede information in all previous revisions/editions of documentation related to Samsung Pay In-App (Online) Payments. Please discontinue use of such previous documentation revisions and copies and refer to this revision/edition for relevant technical guidance.

Table of Contents

List of Acronyms

The following acronyms are commonly used in this document:

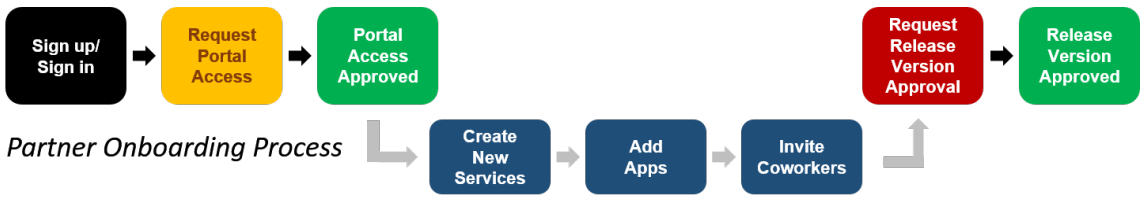
API	Application Programming Interface
APK	Android Application Package
App2App	Application to application integration
ATC	Application Transaction Counter
CGK	Cryptogram Generation Key
CN	Certificate Common Name
CNCC	Card Network Communication Certificate
CSR	Certificate Signing Request
DAC	Discretionary Access Control
DPAN	Digital Personal Account Number | Digital Primary Account Number
E2E	End to end
EMV	Europay, MasterCard, Visa
FCO/ECO	Fast Checkout/Express Checkout
FPAN	Funding Primary Account Number
ID&V / IDV	Identity & Verification
OS	Operating System
OTP	One-time Password
PAN	Primary Account Number | Personal Account Number
PF	Samsung Payment Framework
PID	Partner Identification (string)
PKI	Public Key Infrastructure
POS	Point of Sale
SDK	Software Developer's Kit
TA	Trusted Application
TAV	Token Activation Value
TSP	Token Service Provider
TUI	Trusted User Interface
UUID	Universally Unique Identifier

Welcome Samsung Pay Developers!

The help topics in the pages that follow have been designed to assist you throughout the onboarding, integration, and partner app release approval process. Using the partner portal, you can access valuable resources to help you manage the Samsung Pay features you incorporate into you partner app, including the ability to:

- Create Samsung Pay service groups so you can use different services without having to create multiple accounts
- Invite co-workers to the portal to help you manage Samsung Pay features in your mobile app
- Register your mobile app with Samsung Pay
- Manage your Samsung Pay service(s) and app versions.

The relative timeline for completing partner onboarding is captured in the following diagram.



Incorporating the necessary APIs into your partner app, then testing the integration and validating the results is the final stage before releasing the updated version of your app to the market. Consolidating the whole process down to the most basic level reveals six distinct steps.

This guide takes you through each step in turn. Let's get started.

Step 1. Sign up/Sign in

To sign up as a Samsung Pay partner and request access to the Samsung Pay Developers site, do the following:

a. In your browser, go to https://pay.samsung.com/developers.
b. If you already have a Samsung Account, click **LOG IN** and skip to (e) below; otherwise, click **SIGN UP** and create an account.

   A company business email address that won't change over time is recommended as the primary Samsung Account User ID for managing your portal projects.

c. Agree to the site's terms and conditions and acknowledge that you understand the Samsung Pay Partners Privacy Policy, then click **CREATE A SAMSUNG ACCOUNT**.
d. Fill out the onscreen account creation form, making sure to correctly type the **Security Code**, then click **Continue**.
e. Click **SIGN UP**, enter your Samsung Account **ID** (email address) and **Password**, then click **Sign in**.
f. Look in your email inbox for a Welcome message with an **ACCOUNT ACTIVATION** link and click the link.

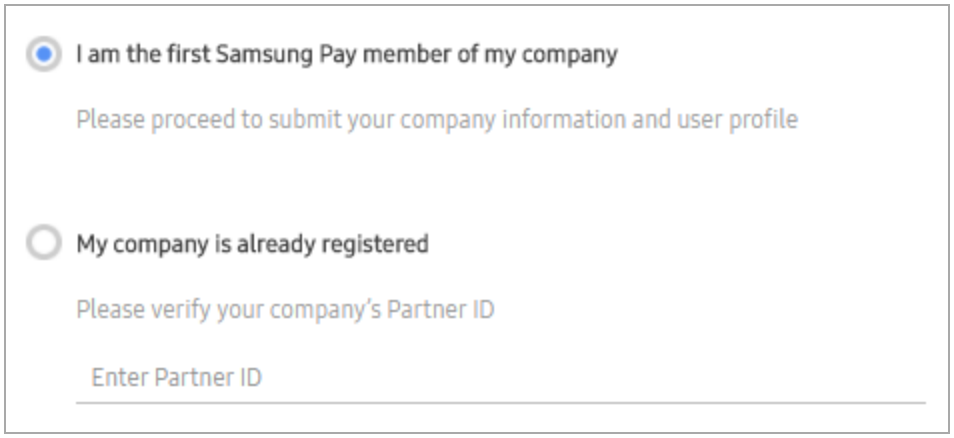
This opens the developers site registration page. Complete the company and user profile described in the next step to become a registered Samsung Pay developer partner.

Step 2. Register your membership

Before you can become a member of the Samsung Pay developer community, we need to know a little bit about you and your company — contact information, type of business, size, etc. This information is entered into a company and user profile, which you can subsequently update as changes occur.

If you're the first one in your company to join Samsung Pay Developers, you will be the principal contact. As such, you'll be given permissions to manage projects and invite others in your company to collaborate. If you're an invited co-worker, you'll need your company's Partner ID to register.

The guidance that follows will help you navigate this process. (*To briefly recap, in Step 1 you created a Samsung Account and received an account activation email, which brought you to the company information and user profile verification page.*)



To register:

1. If you're the first Samsung pay member of your company to register, select the first option. If, on the other hand, you were given a Samsung Pay Partner ID by a co-worker, select the second option — **My company is already registered** — and enter your company's **Partner ID** in the field provided.
2. Click **NEXT**.
3. Complete the company information form and agree to the terms and conditions of use, then click **SAVE AND NEXT**. If you can't complete the profile at this time, click **SKIP AND VERIFY LATER**.
4. Complete the user information form, then click **DONE**; or, if you cannot provide the information at this time, click **SKIP AND VERIFY LATER** .

Upon review of the information provided, your Samsung Pay relationship manager (RM) may request additional details via email. Once your membership registration is approved, you'll be granted access to currently restricted areas of the developers site and you can invite members of your team to collaborate on your Samsung Pay projects. Until then, take advantage of valuable resources like the Samsung Pay SDK and SDK Programming Guide (found under the **RESOURCES** link).

When you receive the email notifying you of membership approval, you're ready to get started. In your browser, return to https://us-partner.pay.samsung.com/ and **SIGN IN**.

In the upper-left corner of the home page, you'll see three menus:

- **GETTING STARTED** – links to quick start guidance on creating and managing services and apps
- **MY PROJECTS** – links to your service management and app management dashboards

- **RESOURCES** – links to download the SDK, its online programming guide, and branding guidelines for proper display of Samsung Pay buttons and logos.

Browse the available **RESOURCES** areas at your leisure, then continue to the next step to learn how to set up and configure your project(s).

Step 3. Set up your partner project

When you integrate your project with Samsung Pay, it's important to have a clear understanding of what's involved. For starters, consider the difference between **apps** and **services**.

In the context of the partner portal, an **app** is your bank's mobile application built on the Android platform, whereas a **service** is a combination of your app, its service type, a CSR, and other parameters that identify your partner app to Samsung Pay. Basically, a service allows you to use the same partner app for multiple purposes.

Issuer app deployment scenario	Unique service-app combinations
Multiple merchant apps using the same PG	Service 1 = <com.merchant.electronicsapp, CSR_PG1> Service 2 = <com.merchant.groceryapp, CSR_PG1>
Multiple web sites using the same PG	Service 1 = <electronicssite.merchant.com, CSR_PG1> Service 2 = <grocerysite.merchant.com, CRS_PG1>
Global merchant app using a different PG for each country	Service 1 = <com.merchant.electronicsapp, CSR_PG1> Service 2 = <com.merchant.electronicsapp, CSR_PG2>
Global merchant web site using a different PG for each country	Service 1 = <electronicsapp.merchant.com, CSR_PG1> Service 2 = <electronicsapp.merchant.com, CSR_PG2>

A **Service ID** (**SID**) represents the unique combination of an app and a **Service Type** (INAPP_PAYMENT) plus a CSR, and is passed as a parameter to Samsung Pay by your partner app in an API call for backend verification and mapping.

Register and manage your apps

To register a partner app:

1. Go to **MY PROJECTS > App management** or click **Services management**in the navigation panel on the left, then click **ADD NEW APP**.
2. Enter the **APP NAME**.
3. Enter its **PACKAGE NAME**, a unique name in reverse domain notation (i.e., *com.myapp.mycompany*) to specifically identify this app.
4. Enter an **APP DESCRIPTION** (optional; helpful for version control).
5. Upload the APK file (optional) by dragging and dropping it in the corresponding box. Click the paperclip icon to browse.
6. Upload representative screenshots (optional) showing UI branding elements/buttons in a supported format (PNG, JPEG, GIF) indicating Samsung Pay as a supported payment option in your app.

   Steps 5 and 6 are required for release versions of your app (see Step 6); not required for initial integration testing.

7. Click **CREATE**, then click **NEXT** at the bottom of the plug-in configuration page.

Your newly added app is now listed in your **App Management**dashboard.

You can view its details by clicking its **APP NAME**. Then, on the **App details** page, you can edit the **APP NAME** and **DESCRIPTION** by clicking the corresponding pencil icon.

Create and manage services

To create a new service:

1. Go to **My Projects > Service management** or click **Services management**in the navigation panel on the left, then click **CREATE NEW SERVICE**.
2. Select **For test** to define the service for initial integration with Samsung Pay, then click **NEXT**.

   You can create a release version of the app after initial integration testing. See Step 6.

3. Select **InApp Online Payment Service (Direct, Indirect)**, then click **NEXT**.
4. Enter the new **SERVICE NAME**.
5. Select **United States** as your **SERVICE COUNTRY**.
6. Select your **PAYMENT GATEWAY** from the list of supported PGs.
7. Drag and drop the **CSR** you share with your PG in the box provided (or click the paperclip to browse).
8. Confirm your agreement with the portal's terms and conditions (click the link to read and print), then click **NEXT**.
9. Select **I already registered app information**, then select the desired **APP NAME** from the list.

   You can add a new app on the fly by selecting **I will upload new app information**.

10. Click **NEXT**.
11. Click **GENERATE NEW KEY** to get a portal-generated **DEBUG API KEY** for preliminary testing of your app with Samsung Pay, then enter up to 10 test accounts (comma-separated). These are Samsung Accounts associated with a registered Samsung Pay app on a supported device.

    You can always edit the service later to add test accounts and/or generate a new **DEBUG API KEY**.

12. Click **DONE** to save the service configuration and see it listed in your **Service management**dashboard.

You can make changes by clicking on the desired **SERVICE NAME** and then clicking the **EDIT INFO** button.

Download the Samsung Pay SDK

To download the SDK:

1. Go to **RESOURCES > SDK download** and select **United States**.
2. Confirm that you agree to the terms of the **Samsung Pay SDK License Agreement** (click the link to read/print).
3. Click **CHECK SDK VERSION** to verify the latest version of the SDK available.
4. Click **DOWNLOAD SDK**and choose an appropriate download location on your computer.
5. Right click the downloaded zip file, select **Extract All...**, then select a Destination Directory.
6. Click **Extract**.

In the destination directory, you'll find three folders:

- **Docs** – contains the official Javadoc API reference and a copy of the Samsung Pay SDK Programming Guide
- **Libs** – contains the SDK jar file and an Apache "Open Source" disclosure
- **Samples** – contains sample apps and the corresponding code demonstrating SDK integration.

Using these resources, you're now ready to integrate your app with the Samsung Pay SDK.

Step 4. Integrate with Samsung Pay

Android Studio is the recommended IDE for doing Samsung Pay SDK integration. For those not familiar with Android Studio, migrating your project requires adapting to a new project structure, build system, and IDE functionality. If you are migrating an Android project from Eclipse, Android Studio furnishes an import tool so you can quickly move existing code into Android Studio projects and Gradle-based build files. See Migrating from Eclipse for more information.

If you are migrating from IntelliJ and your project already uses Gradle, you can simply open your existing project from Android Studio. If you are using IntelliJ but your project does not use Gradle you will need to do a little bit of manual preparation before you can import your project into Android Studio. For more information, see Migrating fro IntelliJ.
Library dependencies in Android Studio use Gradle dependency declarations and Maven dependencies for well-known local source and binary libraries with Maven coordinates. For more information see, Configure Build Variants.

Once you've migrated your project to Android Studio, learn more about building with Gradle and running your app in Android Studio by reviewing Build and Run Your App.

Depending on your project and workflow, you may also wish to read more about using version control, managing dependencies, signing and packaging your app, or configuring and updating Android Studio. If you're new to Android app development, it's wise to spend some time getting acquainted, beginning with a visit to Meet Android Studio.

Otherwise, you're ready to configure your Android project, which includes:

Set up Android Studio

For details on using Android Studio, please see the Android Studio User Guide.

Environment prerequisites

The Samsung Pay SDK requires the Android SDK and the NDK. The Android NDK is an extension of the Android SDK that lets Android developers build performance-critical elements of their apps in native code. The SDK and NDK communicate over the Java Native Interface (JNI). Hence, to set up your development environment, download and install the latest version of the following components in the order listed using the links provided:

- JDK (Java SE) - for installation guidance visit Oracle's Java Platform Installation page
- Android Studio IDE - for installation guidance see Install Android Studio
- Android SDK packages
- Android NDK
- Cygwin environment (optional)

System and device requirements

The Samsung Pay SDK is designed exclusively for Samsung mobile devices supporting Samsung Pay and running Android Lollipop 5.1 (API level 22) or later versions of the Android OS. Use the following code to identify the OS version running on a device to determine whether or not to display the Samsung Pay button in your partner app.

```
Import android.os.Build;
// Check Android version of the device
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.M) {
```

```
// Hide Samsung Pay button
}
```

Add the Samsung Pay SDK to your Android project

Configure Android Studio to include the Samsung Pay SDK in three simple steps:

1. Add **samsungpay.jar** to the **libs** folder of your Android project.
2. Go to **Gradle Scripts > build.gradle** and enter the following dependency:

```
dependencies {
   compile files("libs/samsungpay.jar")
}
```

3. Import the Samsung Pay SDK package into your code.

```
import com.samsung.android.sdk.samsungpay.v2;
```

Next, you're ready to modify your project's manifest file.

Modify your project's manifest file

To appreciate the modifications to the manifest file required to integrate the Samsung Pay SDK, it's important to understand the SDK's **Debug API key** and **API level** attributes. Both are mandatory for placing your app into debug mode initially, and release mode eventually.

Debug API key

The Debug API key is used to verify a validated partner app before it is allowed to interact with the Samsung Pay app. The portal generates the key when you create/edit the service for your app.

The key is valid for three (3) months but may be revoked by Samsung if unresolved issues persist in the partner app.

Partners can request an additional Debug API key after ninety (90) days. Although an unlimited number of devices can use the key, each partner is limited to ten (10) Samsung Accounts for testing purposes.

All partners need to safeguard their **debug_api_key** and **service ID**; these values should never be included in debug logs.

API level

As of Samsung Pay SDK version 1.4, enhanced version control management handles API dependency based on **country** and **Service Type** and improves backward compatibility. This means that, should a partner app be integrated with the latest SDK but continue to use APIs based in a previous level, the implementation remains compatible with Samsung Pay apps supporting the older level without needing to upgrade to the most current version. For example, if the latest SDK has APIs at level 2.0 but the partner app is using APIs at level 1.4, partner app functionality will continue to work with Samsung Pay apps supporting APIs based in level 1.4. Nonetheless, developers are encouraged to keep their partner apps synchronized with the most recent versions of both the Samsung Pay SDK and the Samsung Pay app.

Every API starting from version 1.4 will have an **API level** assigned, and the API reference can be filtered by **API level** to list available classes and APIs accordingly (by API level) when integrating the Samsung Pay SDK. This also means that partner apps must define the API level version in the meta-data of the app's AndroidManifest.xml file. Hence, you can use the **API level** specified in your **AndroidManifest** without upgrading your app to the most recent release of the Samsung Pay SDK.

The earliest supported API level under this scheme is **1.4**.

Usage

The **debug_mode** and **spay_sdk_api_level** custom tags are now mandatory for partner apps running in either debug mode or release mode. Consequently, as of Samsung Pay SDK v1.4, you must specify the applicable API level in your **AndroidManifest** file.

Debug mode

As shown in the snippet below, each partner app running in debug mode must include **debug_mode**, **spay_debug_api_key**, and **spay_sdk_api_level** values as meta-data in the **AndroidManifest** file.

```
<application>
   <meta-data
      android:name="debug_mode"
      android:value="Y" />
   <meta-data
      android:name="spay_debug_api_key"
      android:value="asdfggkndkeie17283094858" />
   <meta-data
      android:name="spay_sdk_api_level"
      android:value="1.7" /> // spay_sdk_api_level -- very important
</application>
```

Be sure to place **<meta-data>** tags inside the **<application>** tag.

Release mode

To validate your app in release mode, the **debug_mode** tag must be set to false (**"N"**) and **spay_sdk_api_level** must be specified.

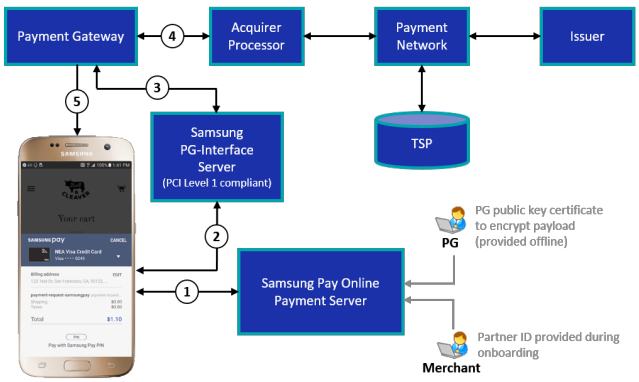
```
<application>
   <meta-data
      android:name="debug_mode"
      android:value="N" />
   <meta-data
      android:name="spay_sdk_api_level"
      android:value="1.7" /> // spay_sdk_api_level -- very important
</application>
```

Again, be sure to place **<meta-data>** tags inside the **<application>** tag.

Next, with your app's manifest properly configured for either debug or release, you're ready to add the necessary API calls for push provisioning. But, first, here's a quick overview of the in-app payment process.

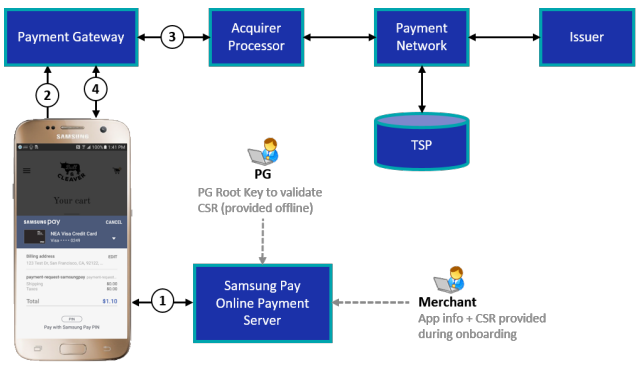
In-App payments overview

Your payment gateway (PG) handles one of two types of tokens — gateway tokens (indirect) or network tokens (direct). Samsung Pay supports requests for both types. So, if you're using a PG like Stripe, you will want to request a *gateway* token from Samsung Pay. But, if you're using a PG like First Data, you'll want to request an encrypted *network* token bundle, which requires you to handle token decryption yourself or work with the PG to handle decrypting the token bundle.



Gateway Token Mode

1. User selects Samsung Pay as the payment method at checkout in the merchant app and the Samsung Pay app requests partner verification from the Samsung Pay Online Payment Server.
2. Encrypted payment information and the **Partner ID** are passed to the Samsung-PG Interface Server.
3. Samsung-PG Interface Server sends a transaction authorization request to the PG on behalf of the merchant; PG authenticates the Partner ID before generating a transaction Reference ID.
4. PG continues payment processing with the acquirer and payment network.
5. The result (approved/declined) is returned to the merchant app on the device for display to the user.



Network Token Mode

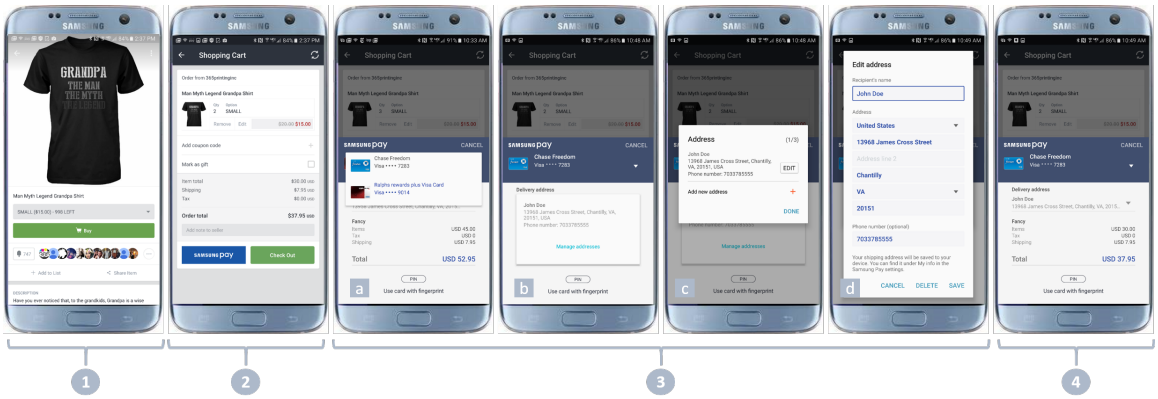
1. User selects Samsung Pay as the payment method at checkout in the merchant app and the Samsung Pay app requests partner verification from the Samsung Pay Online Payment Server.
2. Encrypted payment information is passed from the Samsung Pay app to the PG through the merchant app via the PG SDK.
3. Applying the merchant's private key, PG decrypts the payment information structure and processes the payment through the acquirer and payment network.
4. Upon receiving authorization or rejection, PG notifies the merchant app through its PG SDK.

Check with your PG to determine its specific merchant requirements.

Regardless of the PG model, direct or indirect, the goal is to offer Samsung Pay as a secure payment method from your merchant app. The most common use case involves:

1. User elects to "Buy" and complete the purchase after adding items to a shopping cart.
2. Now in checkout, user selects a checkout option; for example, merchant's "standard" method or Samsung Pay.
3. Upon selecting Samsung Pay, the user is presented with a payment sheet, which includes card selection and shipping address confirmation, with the option to add/modify the "delivery" address for the order, whereupon the user:
    a. makes payment card selection from the list of enrolled cards
    b. chooses to "manage" (change/add) the delivery address
    c. elects to either edit an existing address or add a new address
    d. enters required address information in the form presented and saves it.
4. The user then authenticates the payment method, amount, and delivery address with a fingerprint or PIN.

Your app may be able to support variations of this one-time payment use case for recurring payments and pre-authorized payments using corresponding APIs in your PG's SDK, where available. Consult your PG's merchant developer documentation for additional information.



This general use case captured above takes the following form:

* **TransactionInfoListener** used for normal payment sheet implementations. Substitute **CustomSheetTransactionInfoListener** when implementing Samsung Pay's custom payment sheet.

This requires a number of distinct SDK coding actions:

1. **Create the SamsungPay instance**
2. **Check Samsung Pay app status on the device** (ready/not supported) to determine whether or not to display the Samsung Pay button and/or whether or not to activate/update the Samsung Pay app on the device.
3. **Check/get the list of registered/enrolled cards supported by your merchant app** to determine whether to enable the Samsung Pay button; and, if the button is presented to the user, display only merchant-supported cards for user selection.
4. **Create the transaction request**:
   - request payment with Samsung Pay
   - make transaction details; capture current information from your merchant app
   - refresh transaction details (taxes, shipping fee, total) based on change of delivery address
   - verify delivery address
   - execute payment through your PG

To avoid timing out due to integration with an earlier/deprecated version of the SDK's APIs, be sure to update your partner app with the latest version of the SDK to respond to Samsung Pay callbacks.

Implementing SDK features for In-App Payments

You'll first need to create the **SamsungPay** instance and check the status of the Samsung Pay app on the device. To make the necessary API calls, your partner app must have valid **PartnerInfo** to pass to **SamsungPay** for caller verification.

Create the Samsung Pay instance

Pass a valid **serviceID** and **ServiceType** assigned by the portal during service creation. You must set **ServiceType** in **PartnerInfo** before calling any APIs in order to check for blocked lists and for version control between the Samsung Pay SDK and Samsung Pay app. Be sure, as well, to set the correct API level in your Android manifest.

```
Bundle bundle = new Bundle();

PartnerInfo pInfo = new PartnerInfo(serviceId, bundle);
final SamsungPay samsungPay = new SamsungPay(context, partnerInfo);
```

Here, do not confuse the **serviceID** with the **App ID**; the two are mutually distinct to allow you to register and deploy more than one app in the Samsung Pay ecosystem at a time.

Check the Samsung Pay status

After setting **PartnerInfo**, you can now call **getSamsungPayStatus()**. This method of the **SamsungPay** class must be called before using any other feature in the Samsung Pay SDK.

```
void getSamsungPayStatus(StatusListener callback)
```

The result is delivered to **StatusListener**, which listens for the following events:

- **onSuccess()** – called when the requested operation is successful; provides one of the status codes listed in the table below and extra bundle data related to the request so you can set the appropriate action to take, such as displaying or hiding the Samsung Pay button.
- **onFail()** – called when the requested operation fails; returns an error code and extra bundle data related to the request.

Status Code	Definition
SPAY_READY	Samsung Pay is properly activated and ready to use.
SPAY_NOT_READY	Samsung Pay has not been completely activated, usually because the user did not complete a mandatory update or if the user did not sign in with a valid Samsung Account; in which cases, your partner app can optionally activate or update the Samsung Pay app on the device using the **EXTRA_ERROR_REASON** bundle key
SPAY_NOT_SUPPORTED	Samsung Pay is not supported on this device; typically returned when the device is incompatible with Samsung Pay or the Samsung Pay app is not installed on the device
EXTRA BUNDLE DATA	

Status Code	Definition
EXTRA_COUNTRY_CODE	Current device's ISO 3166-1 alpha 2 country code set by Samsung Pay. Partner app can hide display of the Samsung Pay button based on the resulting country code
EXTRA_ERROR_REASON	Reason for failure set by Samsung Pay
ERROR_SPAY_SETUP_NOT_ COMPLETE	Samsung Pay setup is not complete; tells the partner app to display an appropriate popup message to the user while calling **activateSamsungPay()** to activate the SPAY app
ERROR_SPAY_APP_NEED_TO_ UPDATE	Samsung Pay app version is not current; tells the partner app to display an appropriate popup message to the user and call **goToUpdatePage()**
ERROR_PARTNER_SDK_API_LEVEL	The API level is not valid. Partner app must set valid API level. See Modifying your project's manifest file.
ERROR_PARTNER_SERVICE_TYPE	No **ServiceType** or invalid **ServiceType** set in **PartnerInfo**. See Creating a **SamsungPay** instance.

A **getSamsungPayStatus()** call is structured like this:

```
/*
* Method to get the Samsung Pay status on the device.
* Partner (Issuers, Merchants, Wallet providers, and so on) applications must call this Method to
* check the current status of Samsung Pay before doing any operation.
*/
samsungPay.getSamsungPayStatus(new StatusListener() {
   @Override
   public void onSuccess(int status, Bundle bundle) {
      switch (status) {
         case SamsungPay.SPAY_NOT_SUPPORTED:
            // Samsung Pay is not supported
            samsungPayButton.setVisibility(View.INVISIBLE);
            break;
         case SamsungPay.SPAY_NOT_READY: // Activate Samsung Pay or update Samsung Pay, if needed
            int extra_reason = bundle.getInt(SamsungPay.EXTRA_ERROR_REASON);
            switch(extra_reason) {
               case SamsungPay.ERROR_SPAY_APP_NEED_TO_UPDATE:
                  mSamsungPay.goToUpdatePage();
                  break;
               case SamsungPay.ERROR_SPAY_SETUP_NOT_COMPLETED:
                  mSamsungPay.activateSamsungPay();
                  break;
               default:
                  samsungPayButton.setVisibility(View.INVISIBLE);
                  Log.e(TAG, "Samsung PAY is not ready, extra reason: " + extra_reason);
            }
            samsungPayButton.setVisibility(View.INVISIBLE);
            break;
         case SamsungPay.SPAY_READY:
            // Samsung Pay is ready
            samsungPayButton.setVisibility(View.VISIBLE);
            break;
         default:
            // Not expected result
            samsungPayButton.setVisibility(View.INVISIBLE);
            break;
```

```
        }
    }
    @Override
    public void onFail(int errorCode, Bundle bundle) {
        samsungPayButton.setVisibility(View.INVISIBLE);
        Log.d(TAG, "checkSamsungPayStatus onFail() : " + errorCode);
    }
});
```

Remember, the **getSamsungPayStatus()** method of the **SamsungPay** class must be called before using any other feature of the Samsung Pay SDK. This is because it is your app's only way of determining whether or not the user's device supports Samsung Pay, if your partner app is compatible with the version of Samsung Pay installed, and the ready status of the Samsung Pay app.

If Samsung Pay is supported by the device but not ready (inactive), your app can activate it.

Activate Samsung Pay

If the **status** returned by the check above is **SPAY_NOT_READY**, your app can, as desired or necessary, activate the app on the device and update it to the latest version in release.

The **SamsungPay** class furnishes the following API method:

```
void activateSamsungPay()
```

Call this method to activate the Samsung Pay app on the device running the partner app when the status is **SPAY_NOT_READY** and **EXTRA_ERROR_REASON** is **ERROR_SPAY_SETUP_NOT_COMPLETE**.

Be sure to display an appropriate message alerting the user to sign in upon activation and launch of Samsung Pay.

Here's the code snippet for activation:

```
Bundle bundle = new Bundle();
bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE, SamsungPay.ServiceType.INAPP_PAYMENT.toString());
PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
SamsungPay samsungPay = new SamsungPay(context, partnerInfo);
samsungPay.activateSamsungPay();
```

Update Samsung Pay

The **SamsungPay** class furnishes the following API method to update the Samsung Pay app on a device:

```
void goToUpdatePage()
```

Call this method to activate the Samsung Pay app on the device running the partner app when the status is **SPAY_NOT_READY** and **EXTRA_ERROR_REASON** is **ERROR_SPAY_NEED_TO_UPDATE**.

Be sure to display an appropriate message alerting the user that the Samsung Pay update page will now launch.

Here's the code for launching the update page:

```
Bundle bundle = new Bundle();
bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE, SamsungPay.ServiceType.INAPP_PAYMENT.toString());
PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
SamsungPay samsungPay = new SamsungPay(context, partnerInfo);
```

```
samsungPay.goToUpdatePage();
```

Check/get enrolled cards

Before displaying the Samsung Pay button, your should query the card brand of currently enrolled payment cards to determine if your merchant app supports payment with at lease one enrolled card. For example, if your merchant app accepts one card brand exclusively (e.g., Visa) but the user does not have a Visa card registered in Samsung Pay, you can hide the Samsung Pay button for this purchase at checkout to avoid user confusion. On the other hand, you could just as easily display a message notifying the user: "Sorry, we only accept <*brand*> cards. Please select a different payment method.").

Use the **requestCardInfo()** API method of the **PaymentManager** class but, before calling this method, register **CardInfoListener** so its listener can provide the following events:

- **onResult()** – called when the SDK returns card information from Samsung Pay; result returns information for supported cards or is empty if no card is registered.
- **onFailure()** – called when the query fails; for example, if the SDK service in the Samsung Pay app ends abnormally.

Here's the easiest way to implement a card check and to get cards:

```
Bundle bundle = new Bundle();
bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE, SamsungPay.ServiceType.INAPP_PAYMENT.toString());


PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
paymentManager = new PaymentManager(context, partnerInfo);.
paymentManager.requestCardInfo(new Bundle(), cardInfoListener); // get Card Brand List
//CardInfoListener is for listening requestCardInfo() callback events.
final PaymentManager.CardInfoListener cardInfoListener = new PaymentManager.CardInfoListener() {
   /*
    * This callback is received when the card information is received successfully.
    */
   @Override
   public void onResult(List<CardInfo> cardResponse) {
      int visaCount = 0, mcCount = 0, amexCount = 0, dsCount = 0;
      String brandStrings = "- Card Info : ";

      if (cardResponse != null) {
         PaymentManager.Brand brand;
         for (int i = 0; i < cardResponse.size(); i++) {
            brand = cardResponse.get(i).getBrand();
            switch (brand) {
               case AMERICANEXPRESS:
                  amexCount++;
                  break;
               case MASTERCARD:
                  mcCount++;
                  break;
               case VISA:
                  visaCount++;
                  break;
               case DISCOVER:
                  dsCount++;
                  break;
               default:
                  break;
            }
```

```
            }
        }
        brandStrings += " VI=" + visaCount + ", MC=" + mcCount + ", AX=" + amexCount + ", DS=" +
dsCount;
        Toast.makeText(context, "cardInfoListener onResult" + brandStrings, Toast.LENGTH_LONG).show();
    }

    /*
     * This callback is received when the card information cannot be retrieved.
     * For example, when SDK service in the Samsung Pay app dies abnormally.
     */
    @Override
    public void onFailure(int errorCode, Bundle errorData) {
        // Called when an error occurs during in-app cryptogram generation.
        Toast.makeText(context, "cardInfoListener onFailure : " + errorCode, Toast.LENGTH_LONG).show
();
    }
};
```

Create a transaction request

Upon successful initialization of the **SamsungPay** class, your merchant app is ready to create a transaction request by populating the order details in a Samsung Pay payment sheet, which collects the user's payment information, shipping/billing address(es), and shipping method. For this, Samsung Pay offers two types of online payment sheet — normal and custom.

SDK v1.8.00 does not currently support in-app payments using Discover card. Support for Discover will be included in a subsequent SDK release. In all cases, merchant app developers should update their apps with the latest version of the SDK to respond to Samsung Pay callbacks to avoid timing out using an earlier version of the SDK.

The *normal payment sheet* organizes basic transaction request information in a fixed format for display to the user. The *custom payment sheet* extends the basic format with additional controls to let you configure your payment sheet according to your merchant requirements.

Custom vs. Normal

The normal sheet is managed with the SDK's **PaymentInfo** class, which provides the API tools for standard construction of the payment sheet's address, amount, and payment protocol (3DS, COF, EMV, or OTHER).

The custom sheet is controlled by the **CustomSheetPaymentInfo** class, which replaces the normal sheet's standard attributes with more dynamic controls, including an **AddressControl**for distinguishing between billing and shipping addresses; an **AmountBoxControl**for listing multiple line items, subtotals, and total; a **PlainTextControl** for custom messaging; and a **SpinnerControl** for selection of different shipping methods.

A number of nested classes exist to offer additional flexibility to partner developers. These can be found in the Javadoc SDK-API Reference you downloaded with your SDK package. In this guide we must limit discussion to the most commonly used APIs. Covered next are simplified implementations of both payment sheet types.

Normal payment sheet

In the normal sheet, **PaymentInfo** is sent to Samsung Pay by building [**.build()**] it out for the transaction at hand. Then, upon successful user authentication, Samsung Pay returns the **PaymentInfo** structure and the result string. The result string is forwarded to the PG for transaction completion and will vary based on the specific PG you partner with.

When initializing a payment request, your merchant app must populate the mandatory fields — **merchantName** and **Amount** — in **PaymentInfo**.

PaymentInfo structure

The **PaymentInfo** class is structured like this:

```
public class PaymentInfo implements Parcelable {
// Not populating the mandatory fields will throw an IllegalArgumentException.
   private String version;
   private Amount amount; // mandatory
   private Address shippingAddress;
   private Address billingAddress;
   private String merchantId;
   private String merchantName; // mandatory
   private String orderNumber;
   private PaymentProtocol paymentProtocol;
   private AddressInPaymentSheet addressInPaymentSheet = AddressInPaymentSheet.DO_NOT_SHOW;
   private List<Brand> allowedCardBrand;
   private boolean isCardHolderNameRequired = false;
   private boolean isRecurring = false;
   private String merchantCountryCode;
   private Bundle extraPaymentInfo;
}
```

The corresponding fields and their properties are defined in the following table:

Field	Definition
	MANDATORY
merchantName	Name of merchant as it will appear Samsung Pay payment sheet and on the user's card account statement
amount	Constituate transaction properties (currency, item price, shipping price, tax, total price) which together determine the total amount the user is agreeing to pay the merchant
	OPTIONAL
shippingAddress	Shipping address given to Samsung Pay; related to delivery address
billingAddress	Cardholder's billing address given to Samsung Pay
merchantId	Reference identifier used for a designated purpose at merchant's discretion. If, however, merchant uses the Stripe Payment Gateway or another **indirect PG**, this field is normally set with the PG User ID for the merchant (**Note**: Such an ID is mandatory for merchants working with Stripe)
orderNumber	Created by the merchant app via interaction with a PG. This number is required for refunds and chargebacks. For Visa cards, this value is mandatory. The allowed characters are [A-Z][a-z][0-9,-]; length can be up to 36 characters
paymentProtocol	As specified based on the PG used. The Samsung Pay SDK currently supports 3D Secure for First Data
addressInPaymentSheet	One of six “Address on the Payment Sheet” options regarding billing and/or shipping address. **shippingAddress** is mandatory if **SEND_SHIPPING** or **NEED_BILLING_AND_SEND_SHIPPING** is set for the **addressInPaymentSheet**
allowedCardBrand	Specifies the merchant-supported card brand. If the supported card brand is not specified, all card brands in Samsung Pay can be listed on the Payment Sheet
isCardHolderNameRequired	Determines if the cardholder's name will be displayed on payment sheet

Field	Definition
isRecurring	Depending on the PG and merchant app, recurring payment can be supported; merchant app sets the “Recurring” billing option for cardholder selection and agreement
merchantCountryCode	Merchant's ISO 3166-1 alpha 2 country code
extraPaymentInfo	Additional payment information configured by the merchant app

Before invoking the **startIn-AppPay()** method of **PaymentManager** to request payment with Samsung Pay, you must first pass a valid **PartnerInfo** parcel to **PaymentManager** for caller verification. This is created with **MakeTransactionDetails()**.

```
private PaymentInfo makeTransactionDetails() {
   ArrayList<PaymentManager.Brand> brandList = new ArrayList<>();
   // If the supported card brand is not specified, all card brands in Samsung Pay are
   // listed in the Payment Sheet. Only Visa and Mastercard are currently supported.

   brandList.add(PaymentManager.Brand.MASTERCARD);
   brandList.add(PaymentManager.Brand.VISA);
   // brandList.add(PaymentManager.Brand.AMERICANEXPRESS);
   // brandList.add(PaymentManager.Brand.DISCOVER)

   PaymentInfo.Address shippingAddress = new PaymentInfo.Address.Builder()
      .setAddressee("name")
      .setAddressLine1("addLine1")
      .setAddressLine2("addLine2")
      .setCity("city")
      .setState("state")
      .setCountryCode("United States")
      .setPostalCode("zip")
      .build();

   PaymentInfo.Amount amount = new PaymentInfo.Amount.Builder()
      .setCurrencyCode("USD")
      .setItemTotalPrice("1000")
      .setShippingPrice("10")
      .setTax("50")
      .setTotalPrice("1060")
      .build();

   PaymentInfo.Builder paymentInfoBuilder = new PaymentInfo.Builder()
   PaymentInfo = paymentInfoBuilder
      .setMerchantId("123456")
      .setMerchantName("Sample Merchant")
      .setOrderNumber("AMZ007MAR")
      .setPaymentProtocol(PaymentInfo.PaymentProtocol.PROTOCOL_3DS)
      /* Include NEED_BILLING_SEND_SHIPPING option for AddressInPaymentSheet if merchant needs
       * the billing address from Samsung Pay but wants to send the shipping address to Samsung Pay.
       * Both billing and shipping address will be shown on the payment sheet.
       */
      .setAddressInPaymentSheet(PaymentInfo.AddressInPaymentSheet.NEED_BILLING_SEND_SHIPPING)
      .setShippingAddress(shippingAddress)
      .setAllowedCardBrands(brandList)
      .setCardHolderNameEnabled(true)
      .setRecurringEnabled(false)
      .setAmount(amount)
      .build();
```

```
    return paymentInfo;
}
```

Requesting payment

The methods involved in initiating a payment with Samsung Pay using a normal payment sheet are:

- **startIn-AppPay()** — initiates payment request with Samsung Pay
- **updateAmount()** — updates the transaction amount if either shipping address or card information is changed in Samsung Pay
- **updateAmountFailed()** — returns an error code when the new amount cannot be updated due to a bad address

To request payment, call the **startIn-AppPay()** method with the following data:

PaymentInfo – Displays the online payment sheet of the Samsung Pay app on the merchant app screen. The user can then choose a card for payment from the list of registered cards and/or change the billing and shipping addresses as desired.

TransactionInfoListener – The result of the payment request is delivered here; this listener needs to be registered before calling **startIn-AppPay()**.

The result provides one of the following events:

(a) **onSuccess()** – called when Samsung Pay confirms payment. It provides the **PaymentInfo** object and the **paymentCredential** JSON string. **PaymentInfo** is payment information used for the current transaction containing **amount**, **shippingAddress**, **merchantId**, **merchantName**, **orderNumber**, and **paymentProtocol**. API methods exclusively available in the **onSuccess()** callback comprise:

- **isFastCheckout()** – returns whether or not the Fast Checkout (FCO) option was executed for the transaction (see Appendix A for additional guidance on FCO)
- **getPaymentCardLast4DPAN()** – returns the last 4 digits of the user's digitized personal/primary identification numbers (DPAN)
- **getPaymentCardLast4FPAN()** – returns the last digits of the user's funding personal/primary identification number (FPAN)
- **getPaymentCardBrand()** – returns the brand of the card used for the transaction
- **getPaymentCurrencyCode()** – returns the ISO currency code in which the transaction is valued
- **getPaymentShippingAddress()** – returns the shipping/delivery address for the transaction
- **getPaymentShippingMethod()** – returns the shipping method for the merchandise transacted.

In direct (network token) mode, your merchant app sends a **paymentCredential**, a JSON object containing an encrypted cryptogram. This is passed to the PG. For indirect (gateway token) mode, the JSON object contains a card reference (a token ID generated by the PG) and a status (**AUTHORIZED**, **PENDING**, **CHARGED**, or **REFUNDED**). See Sample paymentCredential in Appendices and Supplementals for example of the credentials returned by direct (nework token) mode (encrypted) and indirect (gateway token) mode.

(b) **onFailure()** – called when a transaction fails; returns the error code and **errorData** bundle for the failure.

Here's how to call the **startIn-AppPay()** method of the **PaymentManager** class to request payment using a normal payment sheet:

```
private void startInAppPay() {
   // PaymentManager.startInAppPay method to show normal payment sheet.
   try {
      Bundle bundle = new Bundle();
      bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE,
         SamsungPay.ServiceType.INAPP_PAYMENT.toString());
      PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
      paymentManager = new PaymentManager(context, partnerInfo);
      /*
       * PaymentManager.startInAppPay is a method to request online(in-app) payment with
```

```
         * Samsung Pay.
         * Merchant app can use this method to make in-app purchase using Samsung Pay from their
         * application with normal payment sheet.
         */
        paymentManager.startInAppPay(makeTransactionDetails(), transactionListener);
    } catch (NullPointerException e) {
         Toast.makeText(context, "All mandatory fields cannot be null.", Toast.LENGTH_LONG).show();
         e.printStackTrace();
    } catch (IllegalStateException e) {
         Toast.makeText(context, "IllegalStateException", Toast.LENGTH_LONG).show();
         e.printStackTrace();
    } catch (NumberFormatException e) {
         Toast.makeText(context, "Amount values is not valid", Toast.LENGTH_LONG).show();
         e.printStackTrace();
    } catch (IllegalArgumentException e) {
         Toast.makeText(context, "PaymentInfo values not valid or all mandatory fields not set.",
             Toast.LENGTH_LONG).show();
         e.printStackTrace();
    }
}

/*
 * TransactionInfoListener is for listening to callback events of online (in-app) payments.
 * This is invoked when card or address is changed by the user on the payment sheet,
 * and also with the success or failure of online (in-app) payment.
 */
private PaymentManager.TransactionInfoListener transactionListener =
    new PaymentManager.TransactionInfoListener() {

    // This callback is received when the user modifies or selects a new address
    // on the payment sheet.
    @Override
    public voidonAddressUpdated(PaymentInfo paymentInfo) {
        try {
          /* Do address verification by merchant app
           * setAddressInPaymentSheet(PaymentInfo.AddressInPaymentSheet.NEED_BILLING_SEND_SHIPPING)
           * If you set NEED_BILLING_SEND_SHIPPING or NEED_BILLING_SPAY with like upper codes,
           * you can get Billing Address with getBillingAddress().
           * If you set NEED_BILLING_AND_SHIPPING or NEED_SHIPPING_SPAY,
           * you can get Shipping Address with getShippingAddress().
           */

          PaymentInfo.Address billing_address = paymentInfo.getBillingAddress();
          int billing_errorCode = validateBillingAddress(billing_address);

          // Call updateAmount() or updateAmountFailed() method. This is mandatory.
          if (billing_errorCode != PaymentManager.ERROR_NONE)
                paymentManager.updateAmountFailed(billing_errorCode);
          else {
                PaymentInfo.Amount amount = new PaymentInfo.Amount.Builder()
                   .setCurrencyCode("USD")
                   .setItemTotalPrice("1000.00")
                   .setShippingPrice("10.00")
                   .setTax("50.00")
                   .setTotalPrice("1060.00")
                   .build();
                paymentManager.updateAmount(amount);
```

```
            }
        } catch (IllegalStateException | NullPointerException e) {
            e.printStackTrace();
        }
    }

    // This callback is received when the user changes the card selected on the payment sheet
    //  in Samsung Pay
    @Override
    public void onCardInfoUpdated(CardInfo selectedCardInfo) {
        /*
         * Called when the user changes card in Samsung Pay. Newly selected cardInfo is passed and
         * merchant app can update transaction amount based on new card (if needed).
         */
        try {
            PaymentInfo.Amount amount = new PaymentInfo.Amount.Builder()
                .setCurrencyCode("USD")
                .setItemTotalPrice("1000.00")
                .setShippingPrice("10.00")
                .setTax("50.00")
                .setTotalPrice("1060.00")
                .build();

            // Call updateAmount() method. This is mandatory.
                paymentManager.updateAmount(amount);
            } catch (IllegalStateException | NullPointerException e) {
                e.printStackTrace();
            }
        }

        /*
         * This callback is received when the online (in-app) payment transaction is approved by
         * user and able to successfully generate in-app payload.
         * The payload could be an encrypted cryptogram (direct in-app payment)
         * or Payment Gateway's token reference ID (indirect in-app payment).
         */
        @Override
        public void  onSuccess(PaymentInfo response, String paymentCredential,
            Bundle extraPaymentData) {
                Toast.makeText(context, "Transaction : onSuccess", Toast.LENGTH_LONG).show();
                // You can use PaymentInfo, paymentCredential and extraPaymentData.
        }

        // This callback is received when the online payment transaction has failed.
        @Override
        public void onFailure(int errorCode, Bundle errorData) {
            Toast.makeText(context, "Transaction : onFailure : "+ errorCode,
                Toast.LENGTH_LONG).show();
        }
    };
```

Call **updateAmountFailed()** if the address provided by Samsung Pay is invalid, out of delivery, or does not exist; For example:

- Merchant does not support the product delivery to the designated location
- Billing address from Samsung Pay is invalid for tax recalculation.

In such cases, the merchant app should call the updateAmountFailed() with one of the following error codes:

- **ERROR_SHIPPING_ADDRESS_INVALID**
- **ERROR_SHIPPING_ADDRESS_UNABLE_TO_SHIP**
- **ERROR_SHIPPING_ADDRESS_NOT_EXIST**
- **ERROR_BILLING_ADDRESS_INVALID**
- **ERROR_BILLING_ADDRESS_NOT_EXIST**

The following example shows you how to use the **updateAmountFailed()** method.

```
// This callback is received when the user modifies or selects a new address on the payment sheet.
@Override
public void onAddressUpdated(PaymentInfo paymentInfo) {
   try {
      // Do address verification by merchant app

      /* setAddressInPaymentSheet(PaymentInfo.AddressInPaymentSheet.NEED_BILLING_SEND_SHIPPING)
       * If you set NEED_BILLING_SEND_SHIPPING or NEED_BILLING_SPAY with like upper codes,
       * you can get Billing Address with getBillingAddress().
       * If you set NEED_BILLING_AND_SHIPPING or NEED_SHIPPING_SPAY,
       * you can get Shipping Address with getShippingAddress().
       */
       PaymentInfo.Address billing_address = paymentInfo.getBillingAddress();
       int billing_errorCode = validateBillingAddress(billing_address);
       if (billing_errorCode != PaymentManager.ERROR_NONE)
          paymentManager.updateAmountFailed(billing_errorCode);
       else {
          PaymentInfo.Amount amount = new PaymentInfo.Amount.Builder()
             .setCurrencyCode("USD")
             .setItemTotalPrice("1000.00")
             .setShippingPrice("10.00")
             .setTax("50.00")
             .setTotalPrice("1060.00")
             .build();
          // Call updateAmount() method. This is mandatory.
          paymentManager.updateAmount(amount);
       }
   } catch (IllegalStateException | NullPointerException e){
      e.printStackTrace();
   }
}
```

The **onAddressUpdated()** callback is available for the normal payment sheet only. For custom payment sheets use **sheetUpdatedListener()** as discussed below.

Custom payment sheet

Many of the same PaymentInfo fields used for the normal payment sheet are included in custom payment sheets. The exception is that the normal sheet's **Amount** and **Address** fields are replaced by **CustomSheet** in **CustomSheetPaymentInfo**.

CustomSheetPaymentInfo structure

To initiate a payment transaction with custom payment sheet, your merchant app should populate the following fields in **CustomSheetPaymentInfo**.

```
public class CustomSheetPaymentInfo implements Parcelable {
```

```
    private String version;
    private String merchantId;
    private String merchantName; //mandatory
    private String orderNumber;
    private PaymentProtocol paymentProtocol;
    private AddressInPaymentSheet addressInPaymentSheet = AddressInPaymentSheet.DO_NOT_SHOW;
    private List<SpaySdk.Brand> allowedCardBrand;
    private CardInfo cardInfo;
    private boolean isCardHolderNameRequired = false;
    private boolean isRecurring = false;
    private String merchantCountryCode;
    private CustomSheet customSheet; //mandatory
    private Bundle extraPaymentInfo;
}
```

The next example shows how to populate **CustomSheet**using **makeCustomSheetPaymentInfo()**.

```
/*
  * Make user's transaction details.
  * The merchant app should send CustomSheetPaymentInfo to Samsung Pay via
  * the applicable Samsung Pay SDK API method for the operation being invoked.
  */
private CustomSheetPaymentInfo makeCustomSheetPaymentInfo() {
   ArrayList<PaymentManager.Brand> bList = new ArrayList<>();
   // If the supported brand is not specified, all card brands in Samsung Pay are
   // listed in the Payment Sheet.
   brandList.add(PaymentManager.Brand.VISA);
   brandList.add(PaymentManager.Brand.MASTERCARD);
   brandList.add(PaymentManager.Brand.AMERICANEXPRESS);

   /*
    * Make SheetControls you want and add to custom sheet.
    * Each SheetControl is located in sequence.
    * There must be a AmountBoxControl and it must be located on last.
    */
   CustomSheet customSheet = new CustomSheet();
   customSheet.addControl(makeBillingAddressControl());
   customSheet.addControl(makeShippingAddressControl());
   customSheet.addControl(makePlainTextControl());
   customSheet.addControl(makeShippingMethodSpinnerControl());
   // customSheet.addControl(makeInstallmentSpinnerControl());
   customSheet.addControl(makeAmountControl());

   CustomSheetPaymentInfo customSheetPaymentInfo = new CustomSheetPaymentInfo.Builder()
      .setMerchantId("123456")
      .setMerchantName("Sample Merchant")
      .setOrderNumber("AMZ007MAR")
      .setPaymentProtocol(CustomSheetPaymentInfo.PaymentProtocol.PROTOCOL_3DS)
      // Merchant requires billing address from Samsung Pay and
      //sends the shipping address to Samsung Pay.
      // Show both billing and shipping address on the payment sheet.
      .setAddressInPaymentSheet(CustomSheetPaymentInfo.AddressInPaymentSheet.
         NEED_BILLING_SEND_SHIPPING)
      .setAllowedCardBrands(bList)
      .setCardHolderNameEnabled(true)
      .setRecurringEnabled(false)
```

```
        .setCustomSheet(customSheet)
        .build();
    return customSheetPaymentInfo;
}
```

Requesting payment

Use the **startInAppPayWithCustomSheet()** method of **PaymentManager** to request payment using a custom payment sheet and **updateSheet()** to revise the custom payment sheet based on user input.

startInAppPayWithCustomSheet() initiates the payment request with a custom payment sheet. After the API is called, the payment sheet persists for 5 minutes. If the timer is expires, the transaction fails.

updateSheet() updates the custom payment sheet if any values on the sheet are changed. As of API level 1.5, a merchant app can also update the custom sheet with a custom error message. See Updating the custom payment sheet with a custom error message in Appendices and Supplementals for additional information.

When you call the **startInAppPayWithCustomSheet()** method, a custom payment sheet is displayed on the merchant app screen. As desired, the user can select a registered card for payment and change the billing and/or shipping addresses. The result is delivered to **CustomSheetTransactionInfoListener**, which provides three events — **onCardInfoUpated()**, **onSuccess()**, and **onFailure()**.

onCardInfoUpdated() is called when the user changes the payment card. In this callback, the **updateSheet()** method must be called to update current payment sheet.

onSuccess() is called when Samsung Pay confirms payment; provides the **CustomSheetPaymentInfo** object and the paymentCredential JSON string, which includes **amount**, **shippingAddress**, **merchantId**, **merchantName**, **orderNumber**, and **paymentProtocol** API methods available exclusively in the **onSuccess()** callback comprise:

- **isFastCheckout()** – returns whether or not the Fast Checkout option was executed for the transaction
- **getPaymentCardLast4DPAN()** – returns the last 4 digits of the user's digitized personal/primary identification number (DPAN)
- **getPaymentCardLast4FPAN()** – returns the last 4 digits of the user's funding personal/primary identification number (FPAN)
- **getPaymentCardBrand()** – returns the brand of the card used for the transaction
- **getPaymentCurrencyCode()** – returns the ISO currency code in which the transaction is valued
- **getPaymentShippingAddress()** – returns the shipping/delivery address for the transaction
- **getPaymentShippingMethod()** – returns the shipping method for the transaction.

In direct (network token) mode, your merchant app sends a **paymentCredential** — JSON object containing an encrypted cryptogram. This is passed to the PG. For indirect (gateway token) mode, the JSON object contains a card reference (a token ID generated by the PG) and a status (**AUTHORIZED**, **PENDING**, **CHARGED**, or **REFUNDED**). See the *Samsung Pay SDK Programming Guide* for examples of the **paymentCredential** returned by direct (network token) mode (encrypted) and indirect (gateway token) mode.

onFailure() is called when the transaction fails; returns the error code and errorData bundle for the failure.

Call startInAppPayWithCustomSheet()

The following snippet shows how to call the **startInAppPayWithCustomSheet()**method.

```
/*
* CustomSheetTransactionInfoListener is for listening callback events of online
* (In-App) custom sheet payment.
* This is invoked when card is changed by the user on the custom payment sheet,
* and also with the success or failure of online (In-App) payment.
*/
private PaymentManager.CustomSheetTransactionInfoListener transactionListener =
    new PaymentManager.CustomSheetTransactionInfoListener() {
```

```
// This callback is received when the user changes card on the custom payment sheet
// in Samsung Pay.
@Override
public void onCardInfoUpdated(CardInfo selectedCardInfo, CustomSheet customSheet) {
   /*
    * Called when the user changes card in Samsung Pay.
    * Newly selected cardInfo is passed and merchant app can update transaction amount
    * based on new card (if needed).
    */
   AmountBoxControl amountBoxControl = (AmountBoxControl)
   customSheet.getSheetControl(AMOUNT_CONTROL_ID);

   // You can set discount rate. Set discount rate for each card brand(optional).
   amountBoxControl.updateValue(PRODUCT_ITEM_ID, 1000);
   amountBoxControl.updateValue(PRODUCT_TAX_ID, 50);
   amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
   amountBoxControl.updateValue(PRODUCT_FUEL_ID, 0, "Pending");
   amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
   customSheet.updateControl(amountBoxControl);
   // Call updateSheet() method. This is mandatory.
   try {
      paymentManager.updateSheet(customSheet);
   } catch (IllegalStateException | NullPointerException e) {
      e.printStackTrace();
   }
}

/*
 * This callback is received when the online (In-App) payment transaction is
 * approved by the user and successfully generates the In-App payload. The
 * payload can be an encrypted cryptogram (direct In-App payment) or the PG's
 * token reference ID (indirect In-App payment).
 */
@Override
public void onSuccess(CustomSheetPaymentInfo response, String paymentCredential,
   Bundle extraPaymentData) {

   /*
    * Called when Samsung Pay successfully creates the In-App cryptogram; Merchant app sends
    * this cryptogram to the merchant server or PG to complete In-App payment
    */
   try {
      String DPAN = response.getCardInfo().getCardMetaData().getString(SpaySdk.EXTRA_LAST4_DPAN,
                        "Null");
      String FPAN = response.getCardInfo().getCardMetaData().getString(SpaySdk.EXTRA_LAST4_FPAN,
         "Null");
         Snackbar.make(fragmentView, " DPAN: " + DPAN + " FPAN: " + FPAN,
            Snackbar.LENGTH_LONG).setAction(getString(R.string.ok), new View.OnClickListener() {
            @Override
            public void onClick(View v) {
            }
          }).show();
      } catch (NullPointerException e) {
            e.printStackTrace();
      }

      Toast.makeText(context, "Transaction : onSuccess", Toast.LENGTH_LONG).show();
```

```
            // You can use PaymentInfo, paymentCredential and extraPaymentData.
        }

        // This callback is received when the online payment transaction has failed.
        @Override
        public void onFailure(int errorCode, Bundle errorData) {
            // Called when some error occurred during In-App cryptogram generation.
            Toast.makeText(context, "Transaction : onFailure : "+ errorCode, Toast.LENGTH_LONG).show();
        }
    };

    private void startInAppPayWithCustomSheet() {
        // PaymentManager.startInAppPayWithCustomSheet to show custom payment sheet.
        try {
            Bundle bundle = new Bundle();
            bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE,
                SamsungPay.ServiceType.INAPP_PAYMENT.toString());
            final PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
            paymentManager = new PaymentManager(context, partnerInfo);
            /*
             * PaymentManager.startInAppPayWithCustomSheet is Method to request online(In-App)
             * payment with Samsung Pay.
             * Merchant app can use this Method to make In-App purchase using Samsung Pay from their
             * application with custom payment sheet.
             */
            paymentManager.startInAppPayWithCustomSheet(makeCustomSheetPaymentInfo(),
                transactionListener);
        } catch (NullPointerException e) {
            Toast.makeText(context, SHORTTAG + "All mandatory fields cannot be null.",
                Toast.LENGTH_LONG).show();
            e.printStackTrace();
        } catch (IllegalStateException e) {
            Toast.makeText(context, SHORTTAG + "IllegalStateException", Toast.LENGTH_LONG).show();
                e.printStackTrace();
        } catch (NumberFormatException e) {
            Toast.makeText(context, SHORTTAG + "Amount values are not valid", Toast.LENGTH_LONG).show();
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
        Toast.makeText(context, SHORTTAG + "PaymentInfo values are not valid or all mandatory fields are
             not set.", Toast.LENGTH_LONG).show();
        e.printStackTrace();
        }
    }
```

Call updateSheet()

Use the **updateSheet()** method when the user updates the address on the custom payment sheet. **errorCode** should be set if the address provided by the Samsung Pay app is invalid, out of delivery, or does not exist. For example:

- Merchant does not support the product delivery to the designated location
- Billing address from Samsung Pay is not valid for tax recalculation

For such cases, the merchant app should call **updateSheet()** with one of the following error codes:

- **ERROR_SHIPPING_ADDRESS_INVALID**
- **ERROR_SHIPPING_ADDRESS_UNABLE_TO_SHIP**
- **ERROR_SHIPPING_ADDRESS_NOT_EXIST**
- **ERROR_BILLING_ADDRESS_INVALID**
- **ERROR_BILLING_ADDRESS_NOT_EXIST**

See Applying the Address Control in a custom payment sheet for guidance on how to use the **updateSheet()** method.

Other custom payment sheet controls include:

- AmountBoxControl – for displaying itemized pricing and total price
- SpinnerControl – for user selection of shipping method and/or installment plans
- PlainTextControl – to display custom messaging on the payment sheet

Each is discussed in Appendices and supplementals or click the desired link above.

Step 5. Test and validate your app

Testing you merchant app is critical to validating Samsung Pay transaction performance to ensure a positive user experience for your users. In truth, the goal of testing is not merely to find errors but to fully understand the quality of your **INAPP_PAYMENT** integration.

Do not conduct release testing in "Debug Mode" (see Usage in Step 4). Debug mode does not accurately reflect how your release version will behave because it bypasses a number of verification steps. Be sure to set "**debug_mode**" to "**N**" (false) in your project's manifest file.

The Samsung Pay partner portal fully supports test versions of your app running alongside Samsung Pay with a valid portal-generated debug-api-key using registered test accounts that you whitelist in the portal.

Testing prerequisites

- One Samsung Pay eligible test device plus one ineligible device, both supporting your partner app to test Samsung Pay app status retrieval accuracy.
- A valid debug-api-key for the service under test. See Step 10 under Create and manage services.
- At least one Samsung Pay test account (up to a total of 10) for devices running your partner app and the Samsung Pay app with a corresponding portal-generated debug-api-key.
- One or more payment cards for transaction testing and to test Check/get enrolled cards for merchant-supported card brands.
- Test/sandbox connectivity with your PG to validate gateway and network token processing of transactions.

Given that all partners are different, with their own personalities and tailored methods, Samsung Pay can only prescribe a general strategy for partner application testing, execution, and management in the form of “Best Practices.” The general guidelines offered here are designed to furnish a baseline for testing and acceptance from which all partners can benefit. Even so, merchant validation checks of Samsung Pay button-display criteria should be conducted in accordance with the guidelines presented below.

Test objectives

The objective of any test is to verify the functionality being examined and validate the result produced. All tests should execute and verify test scripts, identifying, fixing, and then retesting all high and medium severity defects in accordance with specific test criteria. Some basic examples are provided below.

General conditions

The **Samsung Pay button is appropriately shown/hidden as a payment option** according to the following criteria:

Condition/Criteria	Expected Result
Device ineligible	No Samsung Pay button displayed
Country ineligible; not supported by issuer	No Samsung Pay button displayed
Device eligible, Samsung Pay active, eligible payment card present in Samsung Pay	Samsung Pay button displayed
Device eligible, Samsung Pay active, user has no eligible cards in Samsung Pay	No Samsung Pay button displayed

Condition/Criteria	Expected Result
Device eligible, Samsung Pay inactive	Samsung Pay button displayed correctly but not activated; message reads: "Samsung Pay app needs to be set up. Activate the app to complete this purchase with Samsung Pay" or "Please activate the Samsung Pay app to continue."

Recommended test cases (as a minimum)

The following best practices are recommended:

- Test with Samsung Pay app present (installed) on the device, then not installed on device
- Test Samsung Pay app when registered and active with enrolled cards present, then inactive (not registered)
- Test integrated (with Samsung Pay SDK) merchant app on ineligible device
- Test with merchant-supported cards present (enrolled), then with no merchant-supported cards present
- Test transactions with all merchant-supported card brands
- Validate display of applicable normal payment sheet or custom payment sheet
- Test user card selection in payment sheet
- Test with Fast Checkout (FCO) enabled as well as disabled; validate results.
- Test whether partner app accurately determines FCO status.
- Test accuracy of user-entered addresses and changes.
- Test correct display and performance of custom payment sheet controls, when applied.
- Validate all results received from your PG.

Step 6. Release your app

The go-to-market (GTM) process for releasing your app can begin as soon as you upload a release version of your app. After that, keep an eye on the **STATUS** of the release version in *App details* on your portal dashboard — **Debugging** (still in testing), **Pending** (still awaiting approval), **Approved** (certified for release), or **Rejected** (not ready for release).

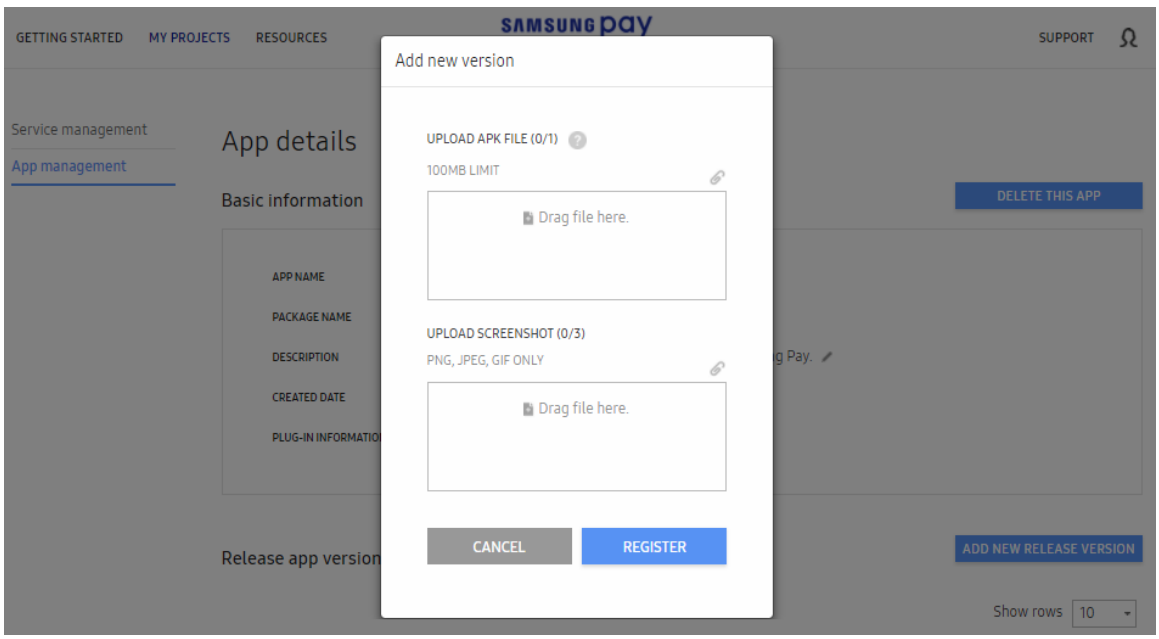
Samsung's approval process is the most stringent for the first (maiden) version of a partner's Samsung Pay SDK-supported release. Subsequent releases may or may not undergo the same degree of vetting.

To begin the release process, upload a new release version of your app:

1. Go to **MY PROJECTS > App management** or click **App management** in the left-hand navigator.
2. Click the **APP NAME** to open its details, then click **ADD NEW RELEASE VERSION**.

> From the **App management** dashboard, you can also click the **ADD NEW VERSION** button corresponding to the **APP NAME** desired.

3. Drag and drop your APK file in the field provided or click the paperclip icon to browse for it.
4. Drag and drop up to three screenshots demonstrating your app's compliance with Samsung Pay's branding guidelines.
5. Click **REGISTER**.



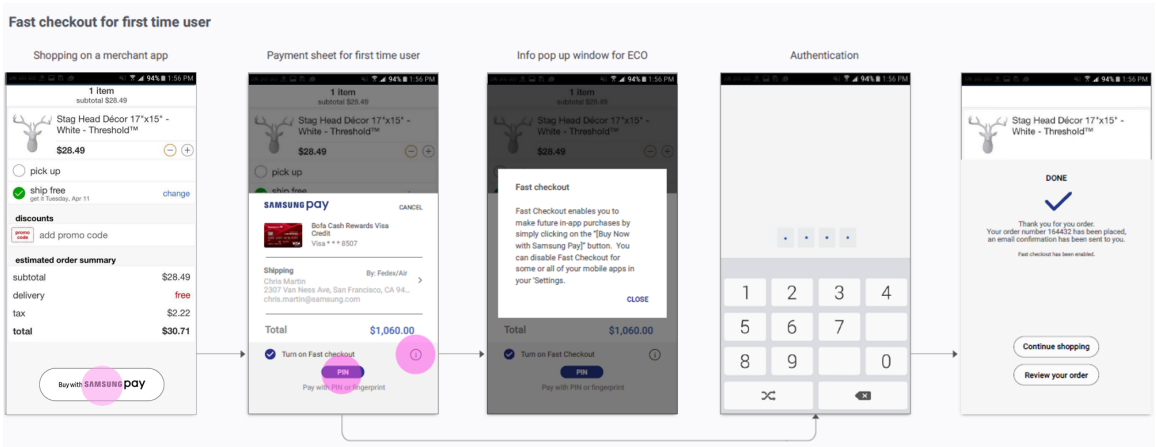
Once the release version is registered, monitor its **STATUS** under **Release app versions**. When approved, you're ready to publish the app to Google Play and Galaxy Apps. If you're not familiar with the app publishing process, visit the respective store website for details.

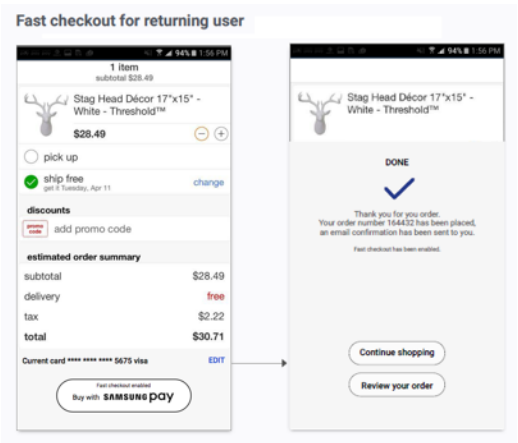
Appendices and Supplementals

Fast Checkout (FCO)

Enabled by default, FCO allows your users to streamline their checkout experience by enabling them to bypass authentication during checkout using their favorite method of payment. When FCO is disabled by the user, the Samsung Pay payment sheet configured for your app, whether normal or custom, is presented to the user as usual. At their option, your users can disable FCO in the Samsung Pay app settings on the device. Once disabled for a particular merchant, FCO can be enabled again by checking the Turn on Fast Checkout control in the merchant's standard payment sheet.

Even when FCO is enabled, however, standard checkout with user authentication via the merchant's normal or custom payment sheet is required the very first time a user makes an in-app purchase using Samsung Pay. Thereafter, if the user has FCO enabled, the last payment card selected, along with the shipping addresses and method used for the last transaction made with Samsung Pay from your app, is stored in the Samsung Pay cloud so it can be used for subsequest transactions without the user having to authenticate.

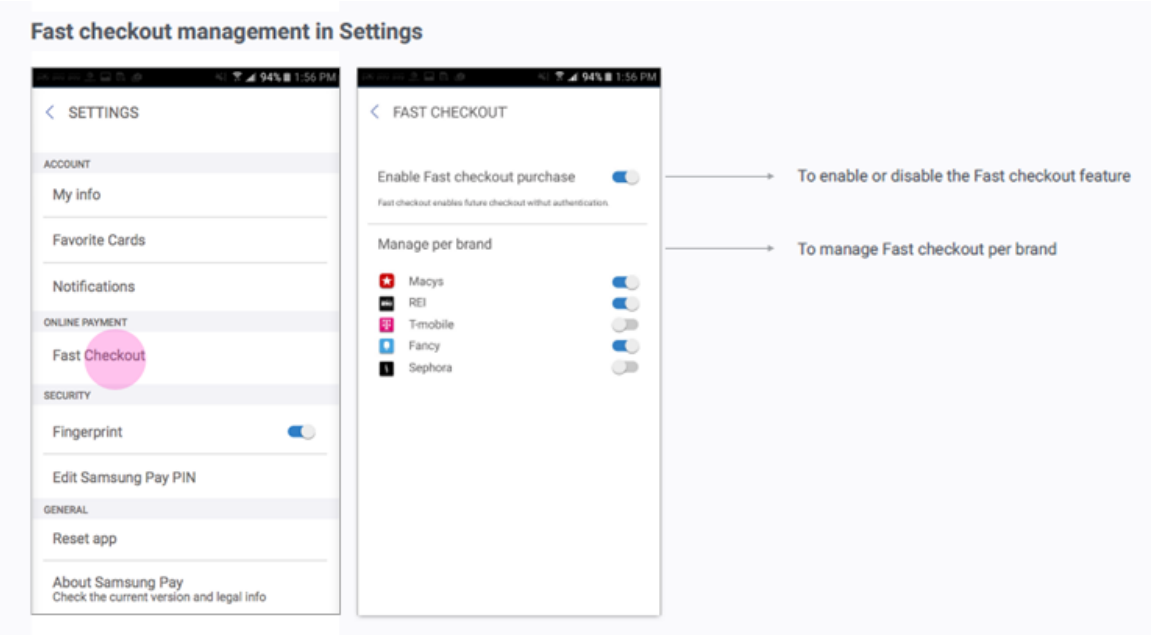




Once enabled, the user can change the registered information used from the last transaction by tapping **EDIT** or **CHANGE** button on the merchant app's checkout page.

If the **EDIT** or **CHANGE** button is not in place on your payment sheet, your merchant app must call the **startInAppPay()** API for normal sheet transactions or **startInAppPayWithCustomSheet ()** for custom payment sheet transaction with the **enableEnforcePaymentSheet** option enabled. Calling the respective APIs with **enableEnforcePaymentSheet** enabled is also required if your PG denies a payload due to a payment card-specific error (e.g., "not enough funds to complete transaction," "low funds" or other failure that results in a rejected transaction by the card issuer).

After enabling FCO, should the user decide to disable the FCO experience and return to the standard payment sheet for purchases, they can do so in the **Settings** menu of Samsung Pay (pictured).



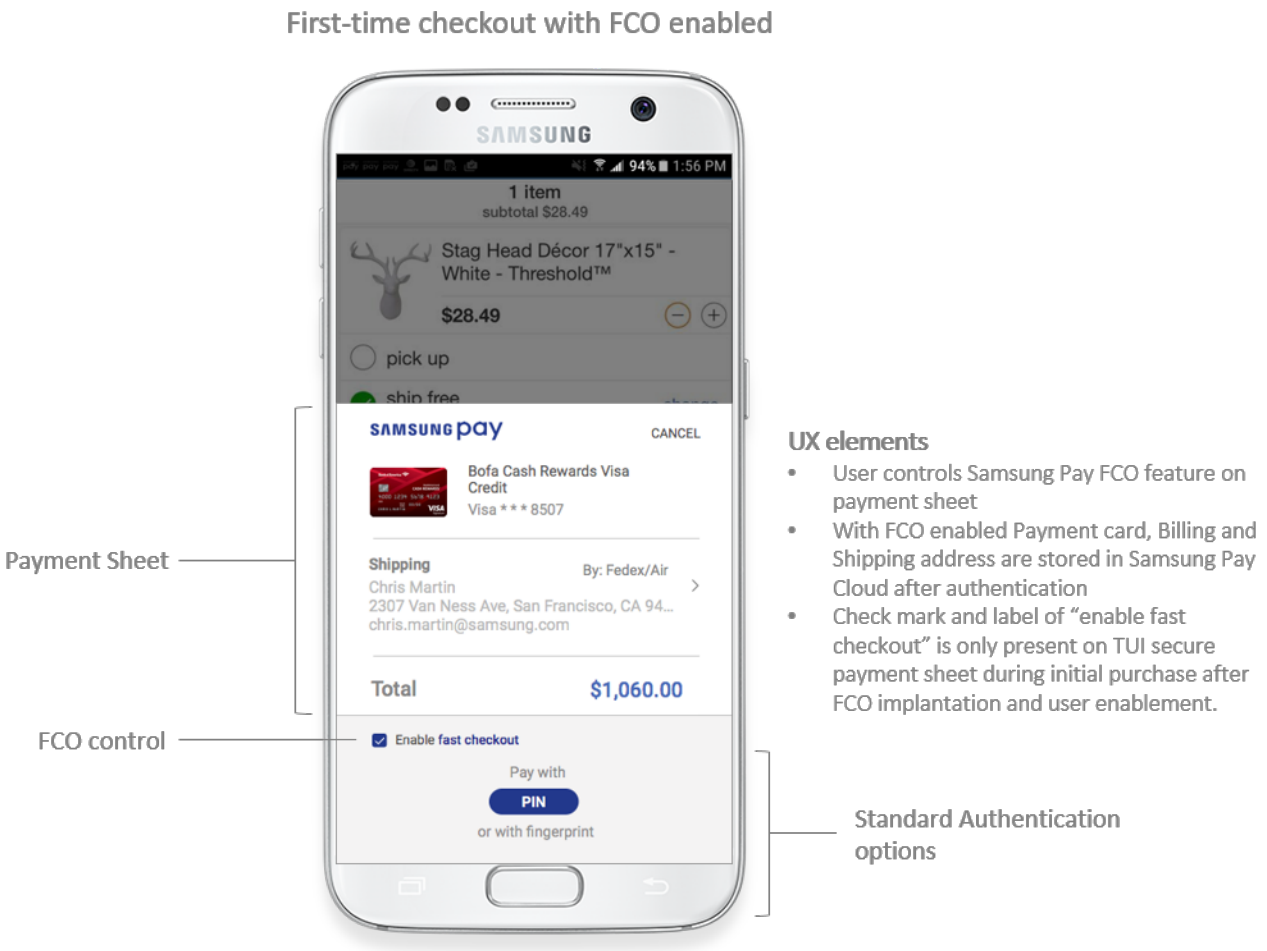
To implement FCO, existing partner apps will need to upgrade to SDK v1.7.00 or a subsequent version and update their Samsung Pay button assets in accordance with the approved UI assets shown above. The key guideline here is that when FCO is enabled, its status is clearly indicated to users so they are not surprised when the transaction goes through without a payment sheet first appearing. A "Fast checkout is enabled" label on or near the Samsung Pay button is recommended to show FCO status. The earliest version of the wallet app that supports external partner app FCO is release 2.8.08.

Branded UI button assets in a variety of forms indicating that FCO is enabled are available on the partner portal.

Initial usage

Upon implementing FCO in a new release of your app, users will encounter your standard payment sheet configuration during their first checkout after upgrading to a version of your app supporting FCO. What will be different is a new **Turn on fast checkout** control in the UI, enabled by default. If the user unchecks the control, FCO is disabled for subsequent transactions unless the user enables it once again from the payment sheet during a subsequent transaction.

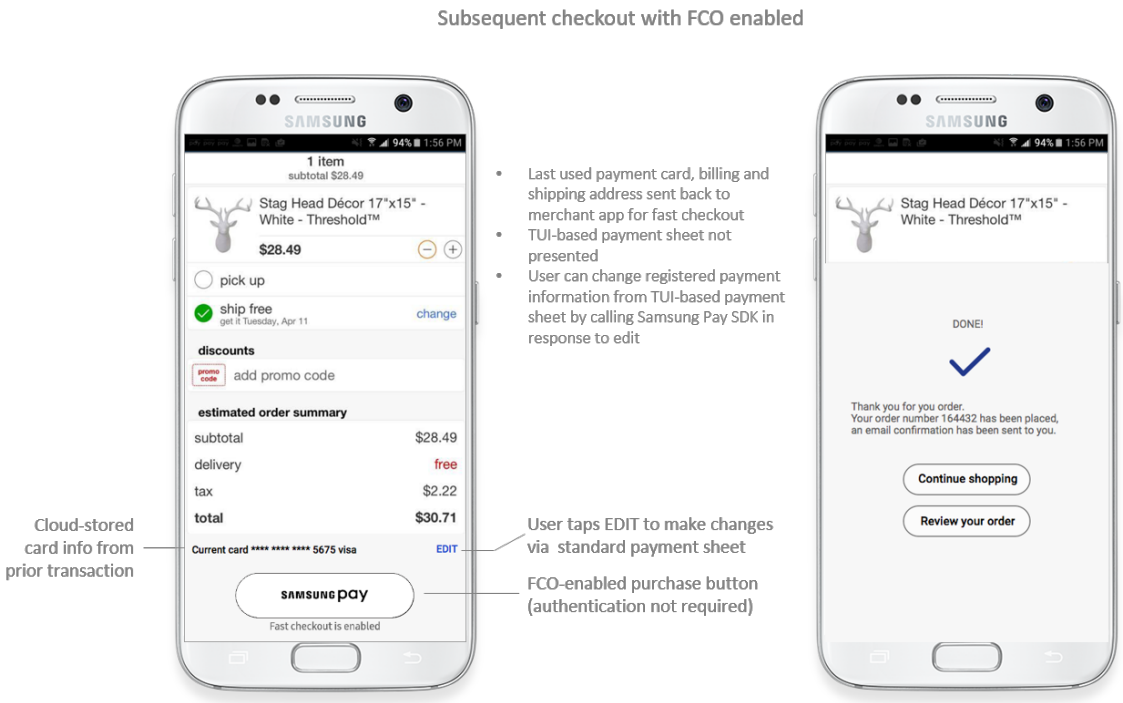
Authentication proceeds as usual for a Samsung Pay in-app transaction —either biometrically or by entering the correct PIN—for this initial purchase.



For subsequent transactions, once the user taps the **Buy with Samsung Pay** button, the payment sheet is not displayed. Instead, the user is taken directly to the order confirmation screen.

For the best user experience, the following changes to your new or existing merchant app are recommended when FCO is enabled:

1. Get the status of FCO after each transaction to ensure that your app knows the status of the FCO option. Save the status of FCO in your app.
2. If FCO is enabled, get the last used payment method and store it.
3. If FCO is enabled, get the last used shipping address and store it.
4. If FCO is enabled display the following:
    a. Last used card (saved from last transaction) with an "**Edit**" option – along with option **enableEnforcePaymentSheet**
    b. Last used shipping address (saved from the last transaction) with an "Edit" option – also with option **enableEnforcePaymentSheet**
    c. Samsung Pay button with the label "**Fast checkout enabled**."



Coding FCO

The next snippet shows you how to configure Samsung Pay buttons and get the last transaction information. Last transaction information includes card brand, currency code, last 4 digits of card DPAN/FPAN, shipping method, and shipping address.

FCO display

```
private boolean isFcoEnabled = false;
private String PaymentCardLast4DPAN = "";
private String PaymentCardLast4FPAN = "";
private SpaySdk.Brand PaymentCardBrand = null;
private CheckBox enforcePaymentSheet_checkbox;

   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
      // ......
      // Fast Checkout view
      enforcePaymentSheet_checkbox=(CheckBox)fragmentView.
      findViewById(R.id.enforce_payment_sheet_checkbox);
      initPayButtonBackground();

      // ......
      return fragmentView;
   }

   /*
    * Initialize Pay button background image
    * Call this function when merchant app is initializing
    */
```

```
    private void initPayButtonBackground() {
       if (null == samsungPayButton) {
          return;
       }
       //Retrieve FCO state which was saved before. Below is a sample code.
       isFcoEnabled = false;
       //Initialize background image according to current FCO state
       if (isFcoEnabled) {
          samsungPayButton.setImageDrawable(getResources().getDrawable(R.drawable.
             fc_rectangular_full_screen_black, null));
       } else {
            samsungPayButton.setImageDrawable(getResources().getDrawable(R.drawable.
               pay_rectangular_full_screen_black, null));
       }
    }
    /*
     * Update Pay button background image
     * Call this function when current transaction finished successfully.
     *
     */
    private void updatePayButtonBackground(boolean isFcoEnabled) {
       if (null == samsungPayButton) {
          return;
       }
       if (isFcoEnabled) {
          samsungPayButton.setImageDrawable(getResources().getDrawable(R.drawable.
             fc_rectangular_full_screen_black, null));
       } else {
          samsungPayButton.setImageDrawable(getResources().getDrawable(R.drawable.
             pay_rectangular_full_screen_black, null));
       }

    }
    //Update PAY button background in CustomSheet after transaction finished successfully
    private PaymentManager.CustomSheetTransactionInfoListener transactionListener =
       new PaymentManager.CustomSheetTransactionInfoListener() {
       // ......
       @Override
       public void onSuccess(CustomSheetPaymentInfo response, String paymentCredential,
          Bundle extraPaymentData) {
          // ......
          // Partner can save this information and if they want, they can use this information
          // for the next transaction without user authentication
          // This sample app uses the information for Fast Checkout control.

          // Get FCO state from response data and update Pay button background accordingly.
          isFcoEnabled = response.isFastCheckout();
          updatePayButtonBackground(isFcoEnabled);

          PaymentCardLast4DPAN = response.getPaymentCardLast4DPAN();
          PaymentCardLast4FPAN = response.getPaymentCardLast4FPAN();
          PaymentCardBrand = response.getPaymentCardBrand();

          // ......
       }
    // ......
```

```
};
```

Samsung Pay FCO currently supports Visa transactions only. FCO support for MasterCard and American Express is coming soon.

Updating the custom payment sheet with a custom error message

An **updateSheet()** method is available for custom error message display on the custom payment sheet.

```
void updateSheet(final CustomSheet sheet, final int errorCode,
   final String customErrorMessage)
```

Pre-defined error messages in earlier versions of the SDK (pre-1.5) have been deprecated.

updateSheet(sheet) can be used to inform the user of any error scenario encountered via a custom message on the custom payment sheet.

```
// In any abnormal cases, merchant can update sheet with CUSTOM_MESSSAGE error code.
paymentManager.updateSheet(customSheet, PaymentManager.CUSTOM_MESSAGE,
   "Phone number is not valid. Please change your phone number!");
```

Sample **paymentCredential**

paymentCredential is the output result of a **startInAppPay()** or **startInAppPayWithCustomSheet()** method.

```
public void onSuccess(PaymentInfo response, String paymentCredential, Bundle extraPaymentData) {
   // You will receive the payloads shown below in paymentCredential parameter
}
```

The **paymentCredential** output structure varies depending on the PG and integration mode (direct – network token, gateway token).

Sample **paymentCredential** output for direct (network token) mode

```
// JWE-only
{
"billing_address":{"city":"BillingCity","country":"USA","state_province":"CA",
"street":"BillingAddr1","zip_postal_code":"123456"}, "card_last4digits":"1122", "3DS" {
"data":"eyJhbGciOiJSU0ExXzUiLCJraWQiOiJCak91a1h2aFV4WU5wOFIwVGs2Y25OaCtZWWFqZXhIeHRVZ0VFdHlhYy9NPSIs
InR5cCI6IkpPU0UiLCJjaGFubmVsU2VjdXJpdHlDb250ZXh0IjoiUlNBX1BLSSIsImVuYyI6IkExMjhHQ00ifQ.Fg2OOUvHdGKkI
VyBa2S5KtUrPWUeujKZEyxz7n6kALhQahszv3P5JaBaOJ-
RoKcznFjDg3qierzjktU7zXST9gwv4Oclahpfdw64w0X6TtAxeYJiIVkJUG-
edXXTWaJeyeIkgC68wEhF1CltSqG4zLWi6upVCAywdPpBN0Hl0C5WcF5Az4WABYtV_
Fda5aHGuyPnE70kEQRTWdlacW9MzEJx2Xth7Msd9OHoulR8LUQ-7gha17jHoOBwgMoQ9q0hAoCNm0LjWiuhKoRyyu-
Njulnbkk8FZus_AIuMgdv2YN9ygFqIlMculb0VWuF0YeKX6IsgAxi0ZQhLiUsJkCZ_w.AuZZxoG46lnrtk3Q.QE2llwS30VzH-
ZduuE8b045CnfRm2p-RjZGBnZcHELS3v26N64cFg1AV5mtP5f-
fSwbJ3ntP5x4V1NK8FmdY0uSPxzeMfvl5badGAC7w9FrXt6X5xV1Fqu6-q-
ZkbxcB9bYgownt983BcKOE1bd5djxFBOdLrc4j68ikDjc5M3LEBDx6hV0aQzKmilCH-
JeVL3AwQyKBny4Vj7m3Fizw7u1PRLI2ZfWUkXDfS4Vwv3bPm4QUDEMVnHXJ.qTYmdmn4ne93juljNmWkJg","type":"S",
```

```
"version":"100"},
"merchant_ref":"MerchantId",
"method":"3DS",
"recurring_payment":false
}
```

You can decrypt the 3DS data block using the merchant's private key. Decrypted output will take the following form:

```
{
"amount":"1000",
"currency_code":"USD",
"utc":"1490266732173",
"eci_indicator":"5",
"tokenPAN":"1234567890123456",
"tokenPanExpiration":"0420",
"cryptogram":"AK+zkbPMCORcABCD3AGRAoACFA=="
}
```

Depending on the structure of the payment processing API provided by your PG, your merchant app can either:

- Pass the entire **paymentCredential** output directly to PG, or
- Extract and pass only the "3DS" part to the PG.

Consult your PG documentation for specific guidance.

The 3DS portion of the **paymentCredential** contais the actual token and cryptogram encrypted in accordance with the specification provided by Samsung (See *Samsung In-App Payment Framework HLD v2.x*)

Sample **paymentCredential** output for indirect (gateway token) mode

For gateway tokens, the **paymentCredential** is the PG's token reference ID with status.

```
{
"reference":"tok_18rje5E6SzUi23f2mEFAkeP7",
"status":"AUTHORIZED"
}
```

In the case of Stripe, your merchant app should be able to pass this token object directly to **Charge** or other appropriate payment processing APIs provided.

Complete specifications for **PaymentManager.TransactionInfoListener.onSuccess** and **PaymentManager.CustomSheetTransactionInfoListener.onSuccess**, respectively, can be found in the Javadoc SDK-API Reference.

Applying the **AddressControl** in a custom payment sheet

The **AddressControl** is used for displaying the billing address or shipping address on a custom payment sheet based on the **My info** menu in the Samsung Pay app. Here, **controlId** and **SheetItemType** are needed to distinguish the billing address from the shipping address.

Your merchant app can set the following properties:

- **AddressTitle** – displays a custom title on the payment sheet. If empty, "Billing address" is display by default.
- **Address** – various methods to return address details, including **phoneNumber** (fetched with the **getPhoneNumber()** method)

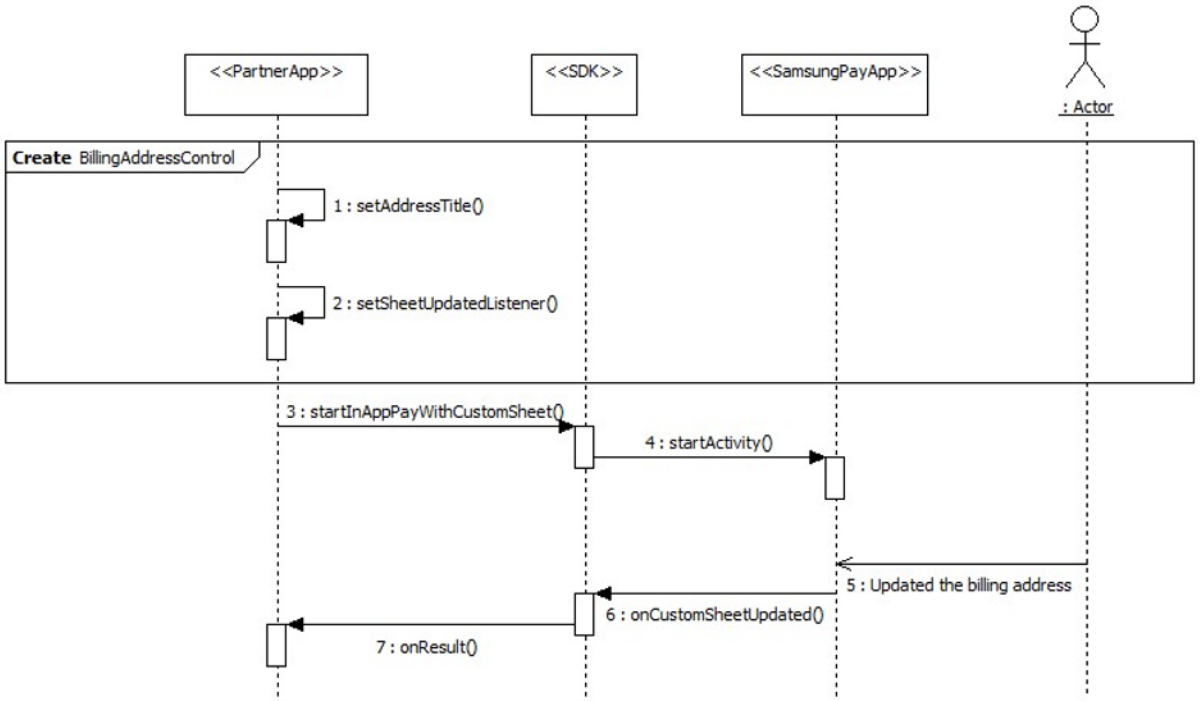
updateSheet()onResult(). Within this callback, the **updateSheet()** method is called to upd.ate the current custom payment sheet.

```
@Override
public void onCardInfoUpdated(CardInfo selectedCardInfo, CustomSheet customSheet) {
   AmountBoxControl amountBoxControl = (AmountBoxControl) customSheet.getSheetControl(AMOUNT_
CONTROL_ID);
   if (amountBoxControl == null) {
      Log.d(TAG, "Transaction : Failed / amountBoxControl is null.");
      return;
   }
   // Now set the updated amount.
   amountBoxControl.updateValue(PRODUCT_ITEM_ID, 1000);
   amountBoxControl.updateValue(PRODUCT_TAX_ID, 50);
   amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
   amountBoxControl.updateValue(PRODUCT_FUEL_ID, 0, "Pending");
   amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
   customSheet.updateControl(amountBoxControl);

   // Call updateSheet() API method with AmountBoxControl. This is mandatory.
   paymentManager.updateSheet(customSheet);
}
```

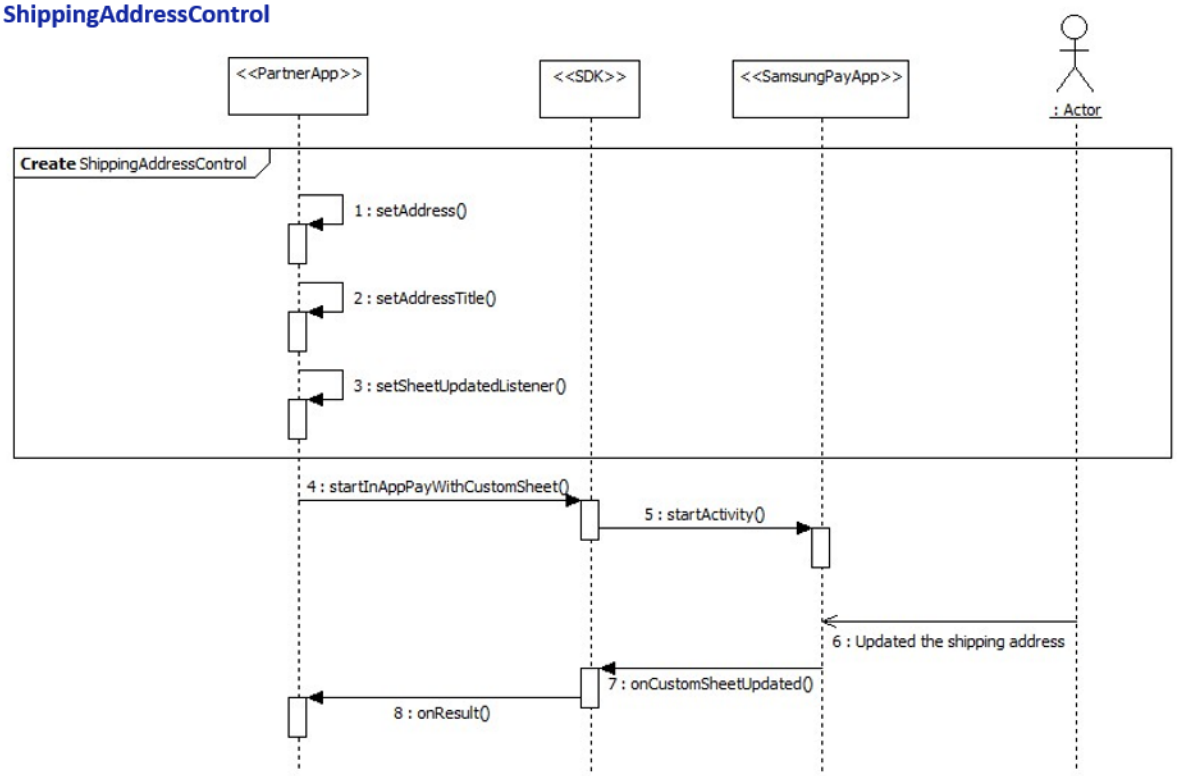
- **ErrorCode** – used for containing error codes directly related to the address. These include:
  – **ERROR_BILLING_ADDRESS_INVALID**
  – **ERROR_BILLING_ADDRESS_NOT_EXIST**
  – **ERROR_SHIPPING_ADDRESS_INVALID**
  – **ERROR_SHIPPING_ADDRESS_UNABLE_TO_SHIP**
  – **ERROR_SHIPPING_ADDRESS_NOT_EXIST**

The workflows for the **BillingAddressControl** and **ShippingAddressControl**are diagrammed below.

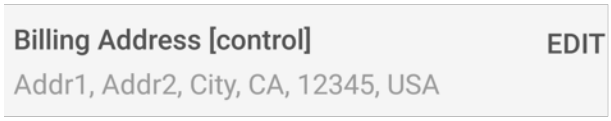
BillingAddressControl

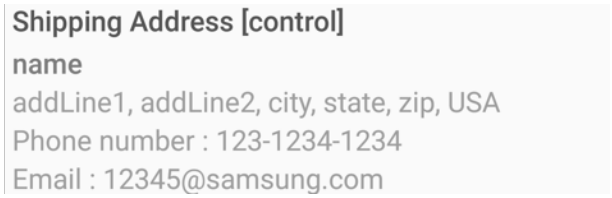


ShippingAddressControl



Here's code samples to display billing and shipping addresses pictured next.






```
// For billing address
private AddressControl makeBillingAddressControl() {
   AddressControl billingAddressControl = new AddressControl(BILLING_ADDRESS_ID,
       SheetItemType.BILLING_ADDRESS);
   billingAddressControl.setAddressTitle("Billing Address [control]");

   //This callback is received when Controls are updated.
   billingAddressControl.setSheetUpdatedListener(new SheetUpdatedListener() {
      @Override
      public void onResult(String updatedControlId, CustomSheet customSheet) {
         Log.d(TAG,"onResult billingAddressControl updatedControlId: " + updatedControlId);

         // Validate billing address and set errorCode if needed.
         AddressControl addressControl =
            (AddressControl) customSheet.getSheetControl(updatedControlId);
         CustomSheetPaymentInfo.Address billAddress = addressControl.getAddress();
         int errorCode = validateBillingAddress(billAddress);
         Log.d(TAG, "onResult updateSheetBilling errorCode: " + errorCode);
         addressControl.setErrorCode(errorCode);
         customSheet.updateControl(addressControl);

         AmountBoxControl amountBoxControl =
            (AmountBoxControl) customSheet.getSheetControl(AMOUNT_CONTROL_ID);
         amountBoxControl.updateValue(PRODUCT_ITEM_ID, 1000);
         amountBoxControl.updateValue(PRODUCT_TAX_ID, 50);
         amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
         amountBoxControl.updateValue(PRODUCT_FUEL_ID, 0, "Pending");
         amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
         customSheet.updateControl(amountBoxControl);

         // Call updateSheet with AmountBoxControl. This is mandatory.
         try {
            paymentManager.updateSheet(customSheet);
         } catch (IllegalStateException | NullPointerException e) {
            e.printStackTrace();
         }
      }
   });

   return billingAddressControl;
}

// For Shipping address
private AddressControl makeShippingAddressControl() {
   AddressControl shippingAddressControl = new AddressControl(SHIPPING_ADDRESS_ID,
      SheetItemType.SHIPPING_ADDRESS);
      shippingAddressControl.setAddressTitle("Shipping Address [control]");
      CustomSheetPaymentInfo.Address shippingAddress = new CustomSheetPaymentInfo.Address.Builder()
         .setAddressee("name")
```

```
            .setAddressLine1("addLine1")
            .setAddressLine2("addLine2")
            .setCity("city")
            .setState("state")
            .setCountryCode("USA")
            .setPostalCode("zip")
            .setPhoneNumber("123-123-1234")
            .setEmail("12345@samsung.com")
            .build();
        shippingAddressControl.setAddress(shippingAddress);
        /*
         * Set address display option on custom payment sheet.
         * If displayOption is not set, default addressControl is displayed.
         * The possible values are any combination of the following constants:
         * {DISPLAY_OPTION_ADDRESSEE}
         * {DISPLAY_OPTION_ADDRESS}
         * {DISPLAY_OPTION_PHONE_NUMBER}
         * {DISPLAY_OPTION_EMAIL}
         */
        int displayOption_val = AddressConstants.DISPLAY_OPTION_ADDRESSEE;
        // Addressee is mandatory
        displayOption_val += AddressConstants.DISPLAY_OPTION_ADDRESS;
        displayOption_val += AddressConstants.DISPLAY_OPTION_PHONE_NUMBER;
        displayOption_val += AddressConstants.DISPLAY_OPTION_EMAIL;
        shippingAddressControl.setDisplayOption(displayOption_val);
    return shippingAddressControl;
}
```

.

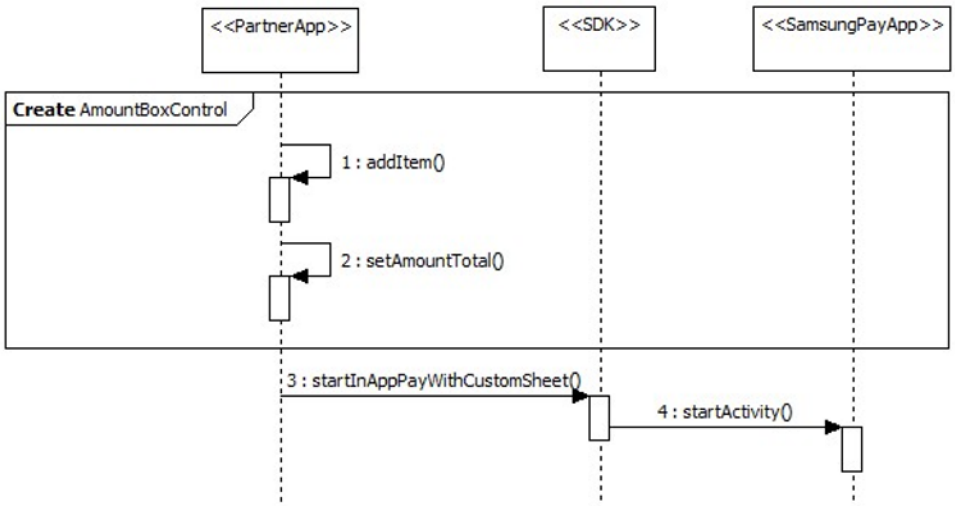
Applying the **AmountBoxControl** in a custom payment sheet

The **AmountBoxControl** is used for displaying purchase amount information. Applying it in a custom payment sheet requires a **controlId** and a **currencyCode** and is populated by **Item**(s) and **AmountTotal**, defined as follows:

- **Item** – consists of **id**, **title**, **price**, and **extraPrice**. When **extraPrice** is included **AmountBoxControl**, its text is displayed; otherwise, **currencyCode** is displayed next to the **price**.
- **AmountTotal** – consists of **price** and **displayOption**, which allows predefined strings only. Your merchant app can set the text to “Estimated amount”, “Amount pending”, “Pending”, “Free”, and so forth. The UI format for the string is different for each option.

The control workflow looks like this:

AmountBoxControl

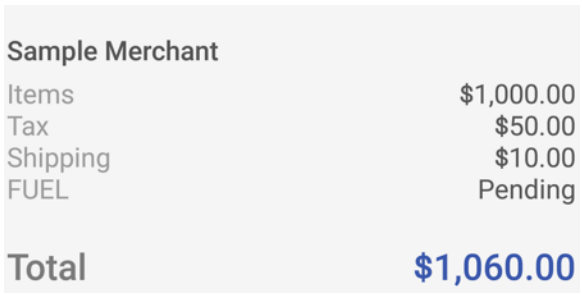


The **setAmountTotal()** API may accept strings that are not predefined as an argument. However, on execution, this will generate an invalid parameter condition or return an error code.

AmountBoxControl usage is demonstrated in the following code snippet:

```
private AmountBoxControl makeAmountControl() {
   AmountBoxControl amountBoxControl = new AmountBoxControl(AMOUNT_CONTROL_ID, "USD");
   amountBoxControl.addItem(PRODUCT_ITEM_ID, "Items", 1000, "");
   amountBoxControl.addItem(PRODUCT_TAX_ID, "Tax", 50, "");
   amountBoxControl.addItem(PRODUCT_SHIPPING_ID, "Shipping", 10, "");
   amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
   amountBoxControl.addItem(3, PRODUCT_FUEL_ID, "FUEL", 0, "Pending");
   return amountBoxControl;
}
```

The resulting custom payment sheet display looks like this:



As seen in the snippet above, your merchant app can add new items using the **addItem()** method during callback. However, it is imperative that the **updateValue()** method be called even if the amount doesn't change.

When the custom sheet is updated, your merchant can add new items to **AmountBoxControl**. For example, let's say a user selects a specific card for which the merchant offers a discount. If so, the discount value item can be added via **updateSheet()**.

```
// Example for adding new item while updating values
AmountBoxControl amount = (AmountBoxControl) sheet.getSheetControll("id_amount");

amount.updateValue("itemId", 900);
amount.updateValue("taxId", 50);
amount.updateValue("shippingId", 10);
amount.updateValue("fuelId", 0);

// Add “Dicount” item
amount.addItem(4, "discountId", "Discount", -60, "");

amount.setAmountTotal(1000, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
sheet.updateControl(amount);

// Call updateSheet with AmountBoxControl. This is mandatory.
try {
   paymentManager.updateSheet(sheet);
} catch (IllegalStateException | NullPointerException e) {
   e.printStackTrace();
}
```

For more specifics and options on custom payment sheet controls, see the Javadoc SDK-API Reference in the **Documentation** folder of your SDK package.

Applying the **SpinnerControl** in custom payment sheet

The **SpinnerControl** is used for displaying spinner options on a custom payment sheet. Applying it requires a **controlId**, **title**, and **SheetItemType** are needed to distinguish between the types of spinner displayed, defined as follows:

- **controlId** – unique value associated with this **SpinnerControl**; used to distinguish it from other **SheetControls**.
- **title** – displays a custom Spinner title on the payment sheet.
- **sheetItemType** – type of sheet item to dispay (**SHIPPING_METHOD_SPINNER** or **INSTALLMENT_SPINNER**).

SHIPPING_METHOD_SPINNER can only be used when the shipping address comes from the Samsung Pay app; i.e., when the **CustomSheetPaymentInfo.AddressInPaymentSheet** option is set to **NEED_BILLING_AND_SHIPPING** or **NEED_SHIPPING_SPAY**.

The merchant app code invoking this class will look something like the following:

```
// Construct Shipping Method SpinnerControl
SpinnerControl spinnerControl = new SpinnerControl(SHIPPINGMETHOD_SPINNER_ID, "Shipping Method ",
    SheetItemType.SHIPPING_METHOD_SPINNER);

// User can select one of these options presented on the payment sheet:
spinnerControl.addItem("shipping_method1", getString(R.string.standard_shipping_free));
spinnerControl.addItem("shipping_method2", getString(R.string.twoday_shipping) );
spinnerControl.addItem("shipping_method3", getString(R.string.oneday_shipping));
spinnerControl.setSelectedItemId("shipping_method1"); // Set default option

// This listen for SheetControl events
spinnerControl.setSheetUpdatedListener(new SheetUpdatedListener() {
   @Override
   public void onResult(String updatedControlId, CustomSheet customSheet) {
```

```
        AmountBoxControl amountBoxControl =
           (AmountBoxControl) customSheet.getSheetControl(AMOUNT_CONTROL_ID);
        SpinnerControl spinnerControl =
           (SpinnerControl) customSheet.getSheetControl(updatedControlId);
        switch (spinnerControl.getSelectedItemId()) {
           case "shipping_method1": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
              break;
           case "shipping_method2": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10 + 0.1);
              break;
           case "shipping_method3": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10 + 0.2);
              break;
           default: amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
               break;
        }

        amountBoxControl.setAmountTotal(1000 + amountBoxControl.getValue(PRODUCT_SHIPPING_ID),
               AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
        customSheet.updateControl(amountBoxControl);

        // Call updateSheet with AmountBoxControl. This is mandatory.
        try {
           paymentManager.updateSheet(customSheet);
        } catch (IllegalStateException | NullPointerException e) {
           e.printStackTrace();
        }
     }
});

// Construct Installment SpinnerControl
SpinnerControl spinnerControl = new SpinnerControl(INSTALLMENT_SPINNER_ID, "Installment",
    SheetItemType.INSTALLMENT_SPINNER);

// User can select one of these options presented on the payment sheet:
spinnerControl.addItem("installment1", "1 month without interest");
spinnerControl.addItem("installment2", "2 months with 2% monthly interest");
spinnerControl.addItem("installment3", "3 months with 2.2% monthly interest");
spinnerControl.setSelectedItemId("installment1"); // Set default option

// Listen for SheetControl events
spinnerControl.setSheetUpdatedListener(new SheetUpdatedListener() {
   @Override
   public void onResult(String updatedControlId, CustomSheet customSheet) {
      AmountBoxControl amountBoxControl =
         (AmountBoxControl) customSheet.getSheetControl(AMOUNT_CONTROL_ID);
      SpinnerControl spinnerControl =
         (SpinnerControl) customSheet.getSheetControl(updatedControlId);
      double TotalInterest = 0;
      switch (spinnerControl.getSelectedItemId()) {
         case "installment1":
            amountBoxControl.updateValue(PRODUCT_TOTAL_INTEREST_ID, TotalInterest);
            break;
         case "installment2": // Calculate total interest again and updateValue
            amountBoxControl.updateValue(PRODUCT_TOTAL_INTEREST_ID, TotalInterest);
            break;
         case "installment3": // Calculate total interest again and updateValue
            amountBoxControl.updateValue(PRODUCT_TOTAL_INTEREST_ID, TotalInterest);
            break;
```

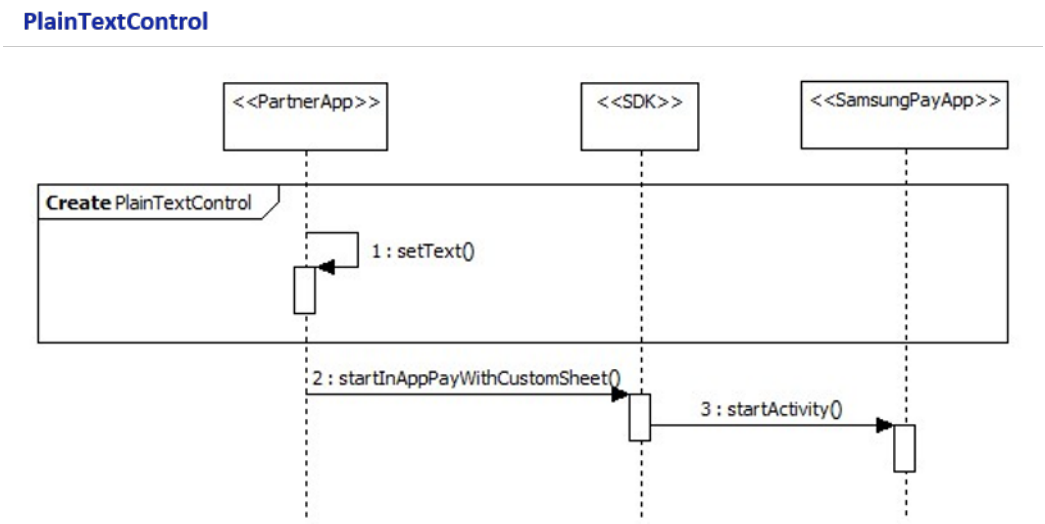
```
            default:
               amountBoxControl.updateValue(PRODUCT_TOTAL_INTEREST_ID, TotalInterest);
               break;
         }

         amountBoxControl.setAmountTotal(1000 + amountBoxControl.getValue(PRODUCT_TOTAL_INTEREST_
            ID), AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
         customSheet.updateControl(amountBoxControl);

         // Call updateSheet with AmountBoxControl. This is mandatory.
         try {
            paymentManager.updateSheet(customSheet);
         } catch (IllegalStateException | NullPointerException e) {
            e.printStackTrace();
         }
      }
   });
```

Again, for additional details and custom sheet control options, see the Javadoc SDK-API Reference in the **Documents** folder of your SDK download package.

Applying the **PlainTextControl** in a custom payment sheet

The **PlainTextControl** is used for displaying a title with a two lines of text or a single line of text without a title. A **controlId** is required. Otherwise, your merchant app sets both the title, as applicable, and the text.

Here's the control flow:



The code is simple:

```
private PlainTextControl makePlainTextControl() {
   PlainTextControl plainTextControl = new PlainTextControl("ExamplePlainTextControlId");
   plainTextControl.setText("Plain Text [example]", "This is example of PlainTextControl");
   return plainTextControl;
}
```

And produces this on the custom payment sheet:

Proguard settings in debug mode

ProGuard is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and methods using short meaningless names.

Here are the Proguard settings for making the Samsung Pay SDK work in debug mode:

```
dontwarn com.samsung.android.sdk.samsungpay.**
-keep class com.samsung.android.sdk.** { *; }
-keep interface com.samsung.android.sdk.** { *; }

-keepresourcexmlelements manifest/application/meta-data@name=spay_sdk_api_level
-keepresourcexmlelements manifest/application/meta-data@name=debug_mode
-keepresourcexmlelements manifest/application/meta-data@name=spay_debug_api_key
```

Setting up staging environment testing

A separate (from PRD) and independent staging environment (STG) is available for partners wishing to test their app before deploying to production a production environment. Partner access to the STG portal is unavailable. Although STG onboarding has the same requirements as PRD onboarding (Steps 1 and 2 above, partners using STG must be manually whitelisted by their Samsung Pay RM, which means providing your RM with the following information:

- Package name (e.g., *com.yourbank.yourapp*)
- Issuer Name(s)
- Up to 10 Samsung Accounts for testing

Given this information, your RM will generate a staging **SID** and **debug-api-key** for your project, which must then be appropriately reflected for STG in your app code, in the main activity and manifest, respectively.

Main Activity

```
    // Service Id from the Samsung Partner Portal - PRD
    public static final String SID_SAMSUNG_PRD = "SAMPLE_PRD_SERVICE_ID";
    // Service Id from the Samsung Partner Portal - STG
    public static final String SID_SAMSUNG_STG = "SAMPLE_STG_SERVICE_ID";
    public static String SID_SAMSUNG = SID_SAMSUNG_PRD;
```

Manifest

```
<application>
   <meta-data
      android:name="debug_mode"
      android:value="Y" />
   <meta-data
      android:name="spay_debug_api_key"
      android:value="asdfggkndkeie17283094858" />
```

```
    <meta-data
       android:name="spay_sdk_api_level"
       android:value="1.7" /> // spay_sdk_api_level -- very important
</application>
```

Issuers configuring their app for STG are reminded that Samsung Pay consists of two APKs bundled together on the device — a user interface (UI) app and a payment framework (PF) app. The latter is strictly the payment engine. Hence, to successfully configure your test devices for staging, both of these APKs must point to STG.

To configure Samsung Pay on your test device(s) for STG:

Navigation paths may vary depending on your device model.

1. Do a factory data reset on the device if the Samsung Pay app is already installed (refer to the Samsung documentation for your device model for details).
2. Install the staging version of the Samsung Pay APK. Contact your RM or product support representative for the file.
3. Copy the debug prop file to the phone's SD card.
4. Go to **Settings** > **Applications** > **Samsung Pay** > **Storage** > **CLEAR DATA** and tap **OK**
5. Enable **Storage** in **Settings** > **Applications** > **Samsung Pay** > **Permissions**
6. Go to **Settings** >**Applications** > **Samsung Pay** and tap **FORCE STOP**.

Next, verify that the UI app is pointing to STG by doing the following:

1. Launch Samsung Pay on the device and sign in.
2. Go to Settings > About Samsung Pay
3. Confirm the STG **debug-api-key** displayed alongside the version number.

Now,verify that the PF app is also pointing to STG:

1. On your PC or Mac, launch a terminal/cmd window and type "**adb shell**" (click here for instructions on installing and using the Android Debug Bridge)
2. Run the following command:

   logcat -v time | grep -i "SpayFw_CommonClient: initializeRequest"

3. Launch Samsung Pay on the mobile device and check the logs on your PC/Mac for an entry like the following:

   09-28 12:14:46.401 15627 17048 I SpayFw_CommonClient: initializeRequest : Environment - STG

About Samsung Electronics Co., Ltd.

Samsung Electronics Co., Ltd. is a global leader in technology, opening new possibilities for people everywhere. Through relentless innovation and discovery, we are transforming the worlds of televisions, smartphones, personal computers, printers, cameras, home appliances, LTE systems, medical devices, semiconductors, and LED solutions. We employ 236,000 people across 79 countries with annual sales exceeding KRW 201 trillion. To discover more, please visit www.samsung.com.

For more information about Samsung Pay, visit http://www.samsung.com/us/samsung-pay/.

Samsung Electronics Co., Ltd.
416, Maetan 3-dong, Yeongtong-gu
Suwon-si, Gyeonggi-do 443-772, Korea

Samsung appreciates all interest and feedback regarding the topics and information in this white paper. Please direct requests for additional information and support-related queries about Samsung Pay security to spay.devhelp@samsung.com.

DISCLAIMER

With respect to this “Samsung Pay Partner Portal Onboarding and Project Integration Guide: In-App Payments for Merchants” and any other documents available from this site or server, neither Samsung nor any of its affiliates or employees makes any warranty, express or implied, including the warranties of merchantability and fitness for a particular purpose, or assumes any legal liability or responsibility for the accuracy, correctness, completeness or usefulness of any information, product, technology or process disclosed.